ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

Titulación :

INGENIERO TÉCNICO EN INFORMÁTICA DE GESTIÓN

Título del proyecto:

APLICACIÓN CON LENGUAJE C# DE ENTRETENIMIENTO BASADO EN UNITY 3D

Autor: Israel Mayayo Peralta

Tutor: Oscar Ardaiz

Pamplona, 27 de Junio de 2011

Índice:

1. Resumen:

En este documento se recoge la memoria del Proyecto de Fin de Carrera para la obtención del título de Ingeniero Técnico Informático de Gestión en la Universidad Pública de Navarra.

Este proyecto que lleva por título "Aplicación con lenguaje C# de Entretenimiento basado en Unity 3D" ha sido realizado por Israel Mayayo Peralta.

El encargado de supervisar el desarrollo del proyecto ha sido Oscar Ardaiz, tutor y profesor en la Universidad Pública de Navarra.

En general, el proyecto consiste en una aplicación de entretenimiento destinada para el uso en iPad, con el cual, el usuario podrá interactuar con la aplicación para conseguir una serie de objetivos. Con esta aplicación se intenta que el usuario pase un rato agradable, además, ha sido construido de manera que su uso sea lo más cómodo e intuitivo.

Para la realización de este proyecto, se han seguido todas las fases del ciclo de vida en un producto: especificación de requisitos, análisis, diseño, implementación y pruebas.

2. Introducción:

2.1.Antecedentes:

Nuestro deseo era construir una aplicación para iPad de entretenimiento, es decir, un juego, por lo que tuvimos que informarnos cuál eran los procesos para conseguir diseñar, implementar y que funcionase una aplicación sobre el iPad.

Para ello, ahora vamos a explicar cuatro aspectos necesarios para la construcción de una aplicación para iPad:

- iPhoneSDK
- Hardware iPad
- Unity 3D
- C Sharp

iPhone SDK:

El lanzamiento kit de desarrollo de software de iPhone fue anunciado en el evento iPhone Software el 6 de marzo de 2008. El SDK permite a los desarrolladores (que utilicen Mac OS X 10.5.4 o superior) crear aplicaciones informáticas usando Xcode que correrán nativamente en el iPhone y en el iPod Touch. Una versión beta fue lanzada después del evento y la versión final fue publicada en julio de 2008 junto con el iPhone

3G. Este evento, acompañado con un gran programa de distribución para terceras personas, se convirtió más tarde en el Programa de Desarrolladores de iPhone que ofrece actualmente cuatro líneas de distribución para los desarrolladores:

- iOS Developer Program
- iOS Enterprise Program
- iOS University Program
- Mac Developer Program

Las aplicaciones distribuidas por medio del programa Developer pueden ser vendidas únicamente a través de la iTunes Store para Mac o para Windows, o bien en la App Store para iPhone o iPod Touch. Los desarrolladores que publiquen sus aplicaciones en la App Store reciben el 70% de los ingresos por ventas y no deben pagar ningún costo de distribución para su aplicación; sin embargo, se requiere pagar una tasa de $99 dólares para utilizar el iPhone SDK y subir aplicaciones a la tienda.

En cambio, las aplicaciones distribuidas por medio del programa de empresas Enterprise son exclusivamente para uso institucional, lo que permite que grandes corporaciones y agencias gubernamentales desarrollen aplicaciones propietarias que no sean para uso público. Este programa genera en el iTunes una especie de "minitienda" sólo accesible por los usuarios corporativos. Tiene un coste de 299$ anuales.

Para que una aplicación funcione en el iPhone, esta necesita estar registrada, lo que certifica que solo pueda ser concedida por Apple, después que el desarrollador haya creado el software por medio del paquete estándar (99 dólares anuales) o del paquete de empresas (299 dólares por año) y del SDK.

Características hardware del equipo:

Hacer juegos para el iPhone / iPad es divertido. Pero para que el juego tenga éxito, es necesario hacer que se ejecute sin problemas. Por lo tanto, es importante comprender las capacidades gráficas y de poder de procesamiento del iPad.

Las características del **iPad** son las siguientes:

- 1 GHz Apple A4 CPU
- Wifi + Bluetooth + (3G Cellular HSDPA, 2G cellular EDGE on the 3G version)
- Accelerometer, ambient light sensor, magnetometer.
- Mechanical keys: Home, sleep, screen rotation lock, volume.
- Screen: 1024x768 pixels, LCD at 132 ppi, LED-backlit.

Formación básica en Unity 3D

Unity 3D es una herramienta de programación integrada para la creación de video juegos 3D u otros contenidos interactivos como visualizaciones arquitectónicas o animaciones 3D en tiempo real. Unity permite crear juegos para las siguientes plataformas: Windows, Mac OS X, Wii, iPhone/iPad, Xbox 360, Android y en breve para Play Station 3.

El desarrollador de este software es Unity Technologies, una empresa con sede en San Francisco y varias oficinas de desarrollo en Dinamarca, Lituania y Reino Unido. Unity Technologies está revolucionando la industria del juego con Unity 3d por su plataforma de desarrollo de gran avance para la creación de juegos 3D, aplicaciones interactivas en 3D, por simulaciones y visualizaciones de formación médica y arquitectónica, en la web, el IOS, Android, y más consolas (Wii y Play Station 3).

Los fundadores de Unity Technologies son David Helgason, Nicholas Francis y Joachim Arte. Utilizan un modelo de negocio que ha revolucionado el modelo de negocios de los juegos. Es gratis para una gran proporción de desarrolladores y asequibles para el resto, con unos ingresos que son los suficientemente fuertes para mantener Unity Technologies como un negocio rentable y que crece rápidamente. Es una empresa privada con el 50 % de los ingresos fuera de EEUU, de los que el 30 % de los ingresos no están relacionados con los juegos.

Sus productos son: Unity 3D, Unity Pro, Asset Server, iOS e iOS Pro.

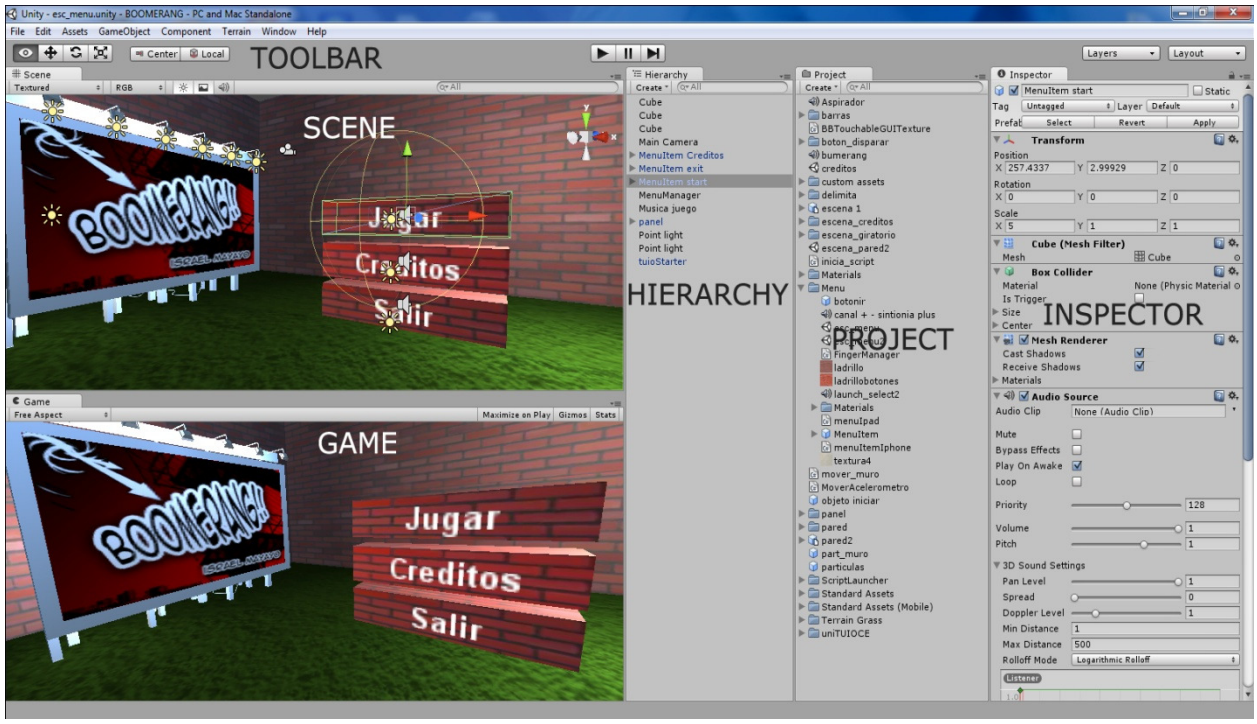
- ***Interfaz Gráfica:***

En este apartado se explicaran la interfaz gráfica para realizar operaciones básicas y las funciones principales para navegar a través de las secciones que Unity ofrece.

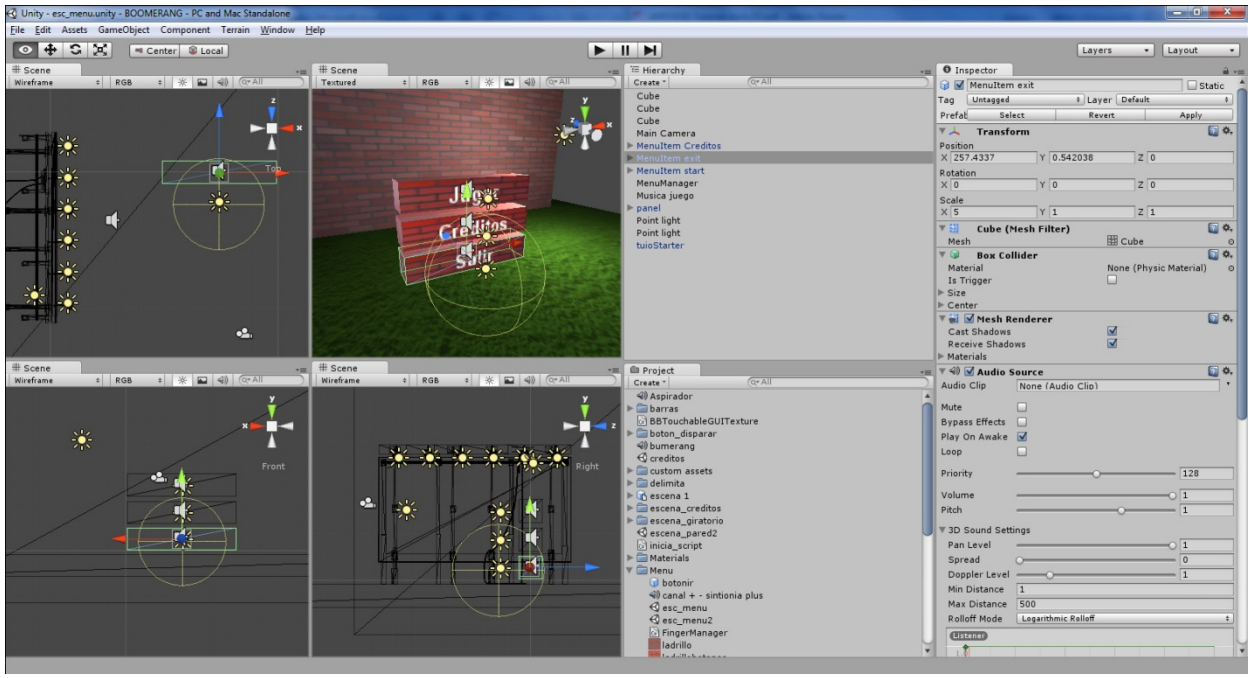
- ***Main Editor Window:***

Es la ventana del editor principal y está compuesta de varias subventanas, llamadas vistas (views). Hay varios tipos de vistas en Unity y cada una de ellos tiene una función específica. La organización de estas vistas en la ventana principal tiene varias configuraciones que puedes modificar con la opción Layout que se encuentra en la parte derecha de Toolbar. Al seleccionar esta opción puedes elegir entre las siguientes opciones:

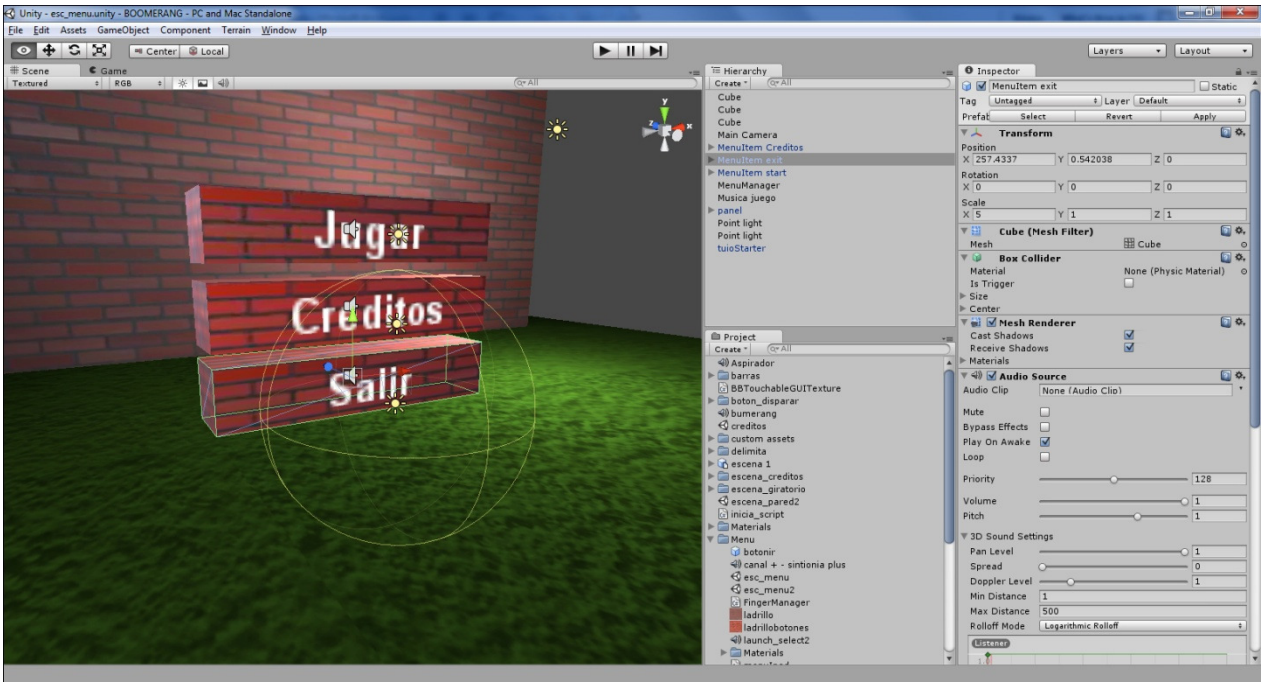
- **2 by 3:** Es la configuración que se observa en la siguiente imagen, consta de 2 (Scene y Game) por 3 (Hierarchy, Project e Inspector).



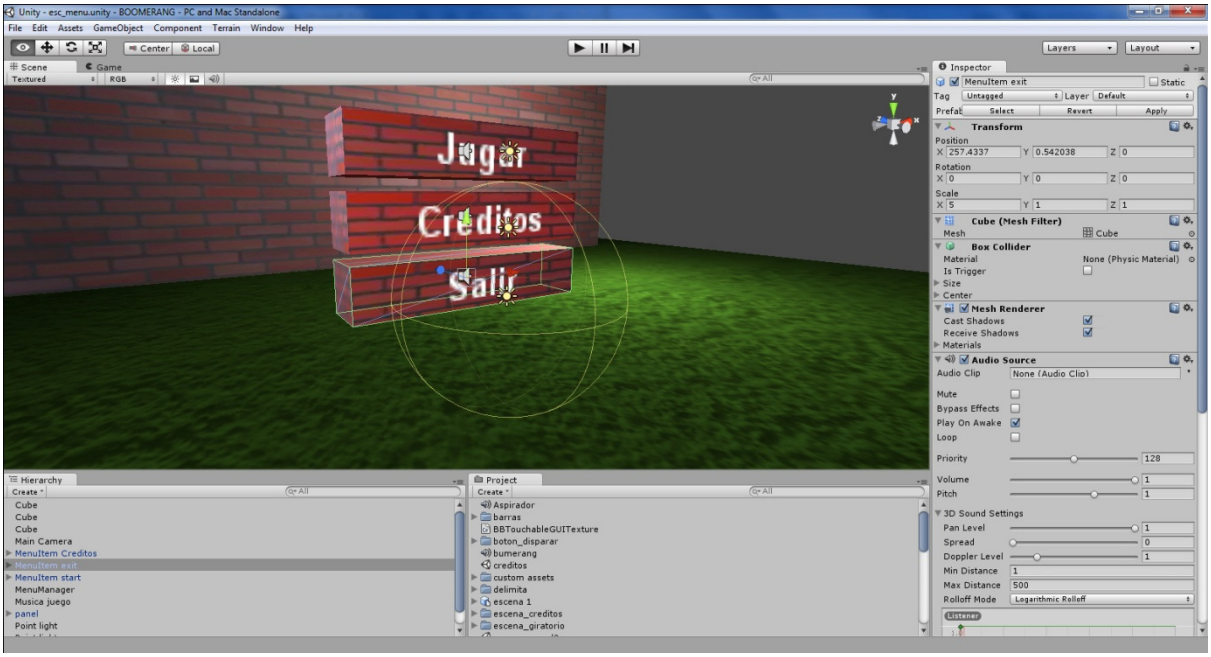
- **4 Split:** Con esta configuración se muestra 4 vistas de Scene pero con diferentes ángulos, y, aparte, las vistas Hierarchy, Project e Inspector. Esta configuración es más apropiada para el modelado en 3D debido a que te da una visión de alzado, perfil y planta de los objetos 3D.



- **Tall:** Esta configuración está compuesta por una vista más grande en la que puedes alternar entre Scene y Game.

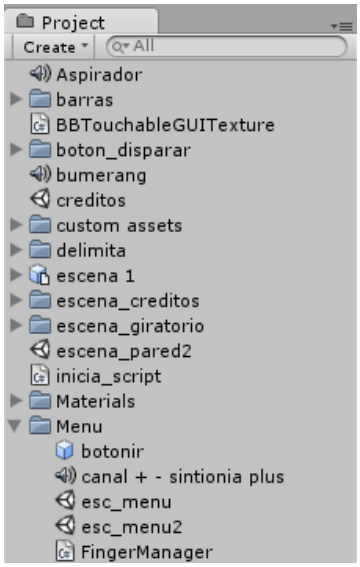


- **Wide:** Es muy similar a la anterior configuración, con la diferencia de la colocación de Hierarchy, Project e Inspector.



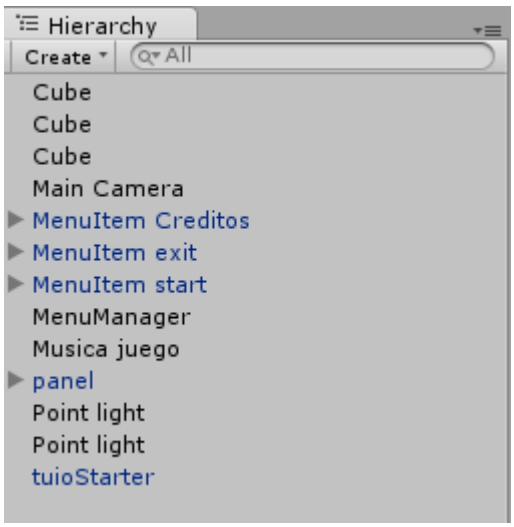
- ***Project View:***

En esta vista puedes encontrar todas las propiedades necesarias para crear tu juego, como scenes, scripts, 3D models, textures, audio files y prefabs. Todo proyecto en Unity tiene una carpeta donde se almacenan todas estas propiedades, también llamadas Assets, pero es muy recomendable no mover Project Assets usando el explorador del sistema operativo ya que se pueden romper vínculos entre datos del proyecto de Unity. Se debe usar siempre el Project View para organizar assets.



- ***Hierarchy:***

Hierarchy contiene en todo momento los GameObjects que se encuentran en escena. Los objetos aparecen y desaparecen de la vista al mismo tiempo que los objetos se eliminen de la escena o se creen nuevos. Algunos de estos objetos son aplicaciones directas de los assets files como 3D models, y otros son aplicaciones de los Prefabs.



- ***Toolbar:***

La barra de herramientas básica consiste en cinco controles principales:

Transform tools:

 Es usada en Scene View para seleccionar, mover, rotar y escalar los objetos del juego.

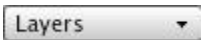
Transform Gizmo Toggles:

Afecta la forma del Scene View.

Control de reproducción:

 Sirve para controlar la reproducción del juego en la vista Game. Puedes reproducir, parar y avanzar en el estado de la aplicación.

Layers Drop-down:



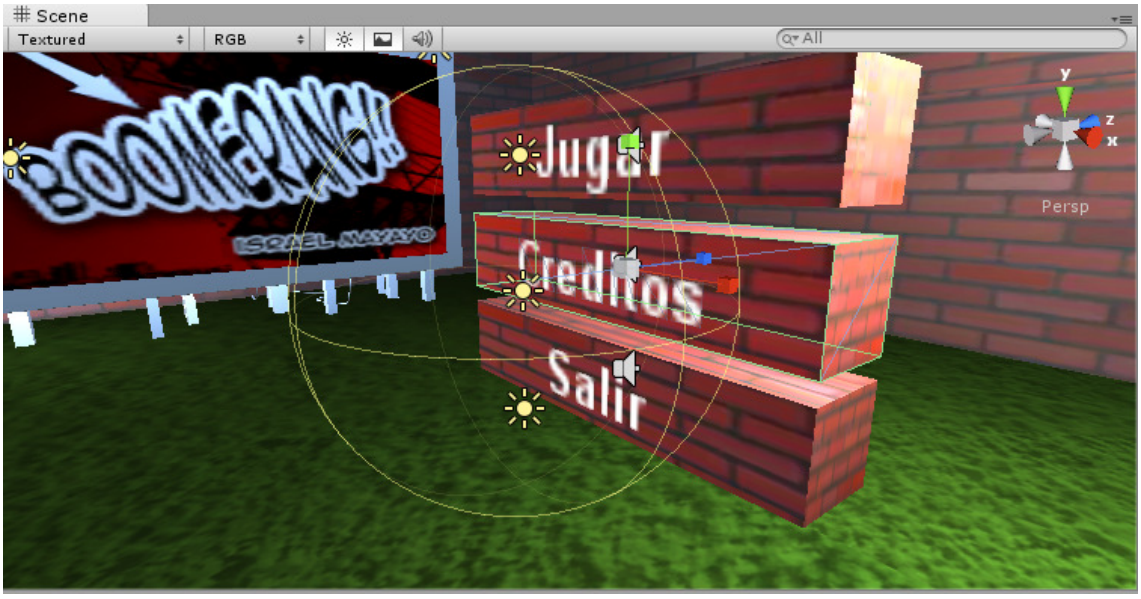
Controla los objetos que son mostrados en la vista Scene.

Layout Drop-down:

Controla la disposición de las vistas en la ventana principal como ya hemos nombrado anteriormente.

- ***Scene View:***

La vista Scene es una ventana interactiva en la cual puede seleccionar y posicionar los ambientes, el jugador, la cámara, y todos los demás GameObjects que intervengan en la aplicación. Para realizar todas estas funciones sobre Scene View debe utilizar Transform tools y el ratón del ordenador.



Utilizando la herramienta “mano” podrá mover la cámara de Scene View de izquierda a derecha, de arriba abajo pulsando el botón izquierdo del ratón. En caso de que no tener seleccionada la opción mano, obtendrá la misma función presionando la rueda del ratón.

Manteniendo **Alt** y haciendo click puede girar sobre un punto central para obtener otro punto de vista.

Si pulsas sobre la vista Scene con el botón derecho del ratón y arrastras, la cámara Scene gira pero sobre el punto donde este situada la cámara Scene.

Para hacer **Zoom** en la vista Scene tiene dos posibilidades, o bien con la rueda del raton, o pulsando **Alt** y botón derecho del ratón y arrastrando.

Otra función interesante, es la utilización de la tecla mayúscula (shift) al mismo tiempo que las anteriores combinaciones para que el efecto de Zoom o desplazamiento de la cámara sea mucho más rápido.

También es posible cambiar la cámara de la vista Scene a modo isométrico seleccionando el punto central o perspectiva, u obtener otro ángulo pulsando cualquiera de los ejes (X, Y, Z).

Otra manera de seleccionar las opciones de **Transform tools** es a través de los atajos de teclado. Las teclas Q, W, E, R corresponden a las herramientas hand, translate, rotate y scale respectivamente.

En el proceso de construcción de los juegos, se tendrá que ubicar varios objetos. Para ello, debemos usar **Transform Tools** en el **toolbar** para trasladar, rotar y escalar a GameObjects. Cada una de estas herramientas posee un **Gizmo** que aparece alrededor del objeto que esta seleccionado en el **Scene View**. Se puede usar el ratón y manipular cualquier **Gizmo** axis para modificar el componente Transform del GameObject, o también se pueden introducir valores directamente al componente Transform que se encuentra en el **Inspector.**

La vista Scene contiene una barra de control que nos permite ver la escena en varios modos de vista como textured, wireframe, RGB, overdraw, y otros. Esta barra te permitirá ver iluminaciones, oír los objetos con sonido, y elementos del juego.

- ***Game View:***

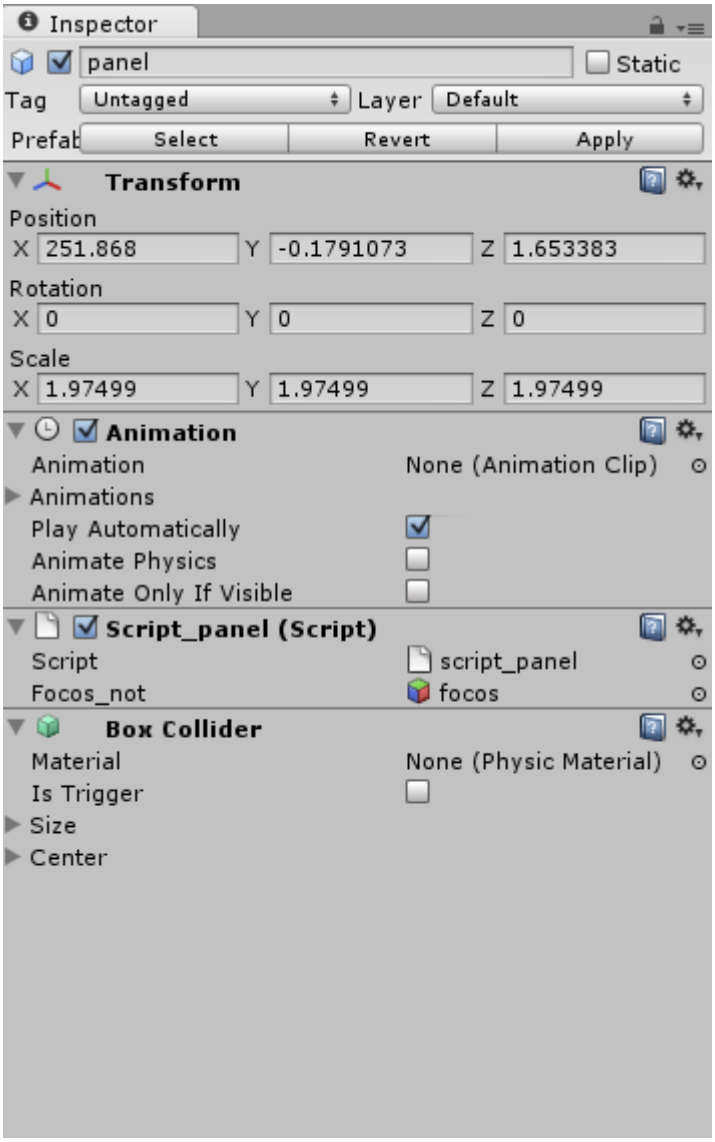
La vista Game es generada desde la Main Camera, es decir, la cámara o las cámaras del juego. Es una vista representativa en la cual se puede observar el producto final sin tener que construir el proyecto y ejecutarlo.

La manera de controlar la reproducción del juego sobre esta vista es mediante la barra de control de reproducción de **Toolbar.**  Estos botones se utilizan para controlar el editor Play Mode, para así, observar cómo funciona el producto final. Mientras se está ejecutando puedes observar en las demás vistas cómo evoluciona la aplicación y puedes realizar modificaciones sobre los objetos del juego ya que una vez parada la reproducción estos valores serán inicializados de nuevo con el valor antiguo.

Mediante la barra de control de la vista Game puedes controlar las dimensiones del Game View con Free Aspect, maximizar al 100% la vista en modo Play, observar los Gizmos durante se este reproduciendo y mostrar las estadísticas de renderizado, que es muy útil para modificar y optimizar los gráficos.

- ***Inspector:***

Los juegos y otras aplicaciones creadas por Unity están compuestos por varios GameObjects que contienen meshs (mallas), scripts, sounds, colliders y otros componentes. En la vista Inspector se muestra todos estos componentes que contienen el objeto seleccionado y todas sus propiedades.



De este modo, cualquier propiedad de los componentes de los objetos puede ser modificada directamente sobre el **Inspector** sin cambiar el script. Otra utilidad del **Inspector** es cambiar valores de las variables durante el **Modo Play** para realizar pruebas.

La opción más importante, en mi opinión, es poder referenciar objetos a variables solo arrastrando. Es decir, puedes definir una variable en un Script como pública del tipo **Transform** o **GameObject** por ejemplo, y luego arrastrar y soltar sobre la variable en la vista **inspector** para así asignarlo como variable del Script.

Formación sobre el lenguaje C Sharp:

Vamos a explicar las características del lenguaje, realizando primero un recorrido por su historia, ver su creación, y, finalmente, comparando el nuevo lenguaje con Java, el lenguaje de programación orientado a objetos por excelencia.

- ***Historia de C, C++ y C#:***

El lenguaje de programación C# fue creado con el mismo espíritu que los lenguajes C y C++. Esto explica sus poderosas prestaciones y su fácil curva de aprendizaje. No se puede decir lo mismo de C y C++, pero como C# fue creado desde cero. Microsoft se tomó la libertad de eliminar algunas de las prestaciones más pesadas (como los punteros). Esta sección echa un vistazo a los lenguajes C y C++, siguiendo su evolución hasta C#.

El lenguaje de programación C fue diseñado en un principio para ser usado en el sistema operativo UNIX. C se usó para crear muchas aplicaciones UNIX. incluyendo un compilador de C. y a la larga se usó para describir el mismo UNIX. Su amplia aceptación en el mundo académico se amplió al mundo comercial y los proveedores de software como Microsoft y Borland publicaron compiladores C para los ordenadores personales. El API original para Windows fue diseñado para trabajar con código Windows escrito en C y el último conjunto de API básicos del sistema operativo Windows sigue siendo compatible con C hoy en día.

Desde el punto de vista del diseño. C carecía de un detalle que ya ofrecían otros lenguajes como Smalltalk: el concepto de objeto. Piense en un objeto como en una colección de datos y un conjunto de operaciones que pueden ser realizadas sobre esos datos. La codificación con objetos se puede lograr usando C pero la noción de objeto no era obligatoria para el lenguaje. Si quería estructurar su código para que simulara un objeto, perfecto. Si no, perfecto también. En realidad a C no le importaba. Los objetos no eran una parte fundamental del lenguaje, por lo que mucha gente no prestó mucha atención a este estándar de programación.

Una vez que el concepto de orientación a objetos empezó a ganar aceptación, se hizo evidente que C necesitaba ser depurado para adoptar este nuevo modo de considerar al código. C++ fue creado para encarnar esta depuración. Fue diseñado para ser compatible con el anterior C (de manera que todos los programas escritos en C pudieran ser también programas C++ y pudieran ser compilados con el compilador de C++). La principal aportación a C++ fue la compatibilidad para el nuevo concepto de objeto. C++ incorporó compatibilidad para clases (que son “plantillas” de objetos) y permitió que toda una generación de programadores de C pensaran en términos de objetos y su comportamiento.

El lenguaje C++ es una mejora de C, pero aun así presenta algunas desventajas. C y C++ pueden ser difíciles de manejar. A diferencia de lenguajes fáciles de usar como Visual Basic, C y C++ son lenguajes de muy "bajo nivel" y exigen de mucho código para funcionar correctamente. Tiene que escribir su propio código para manejar aspectos como la gestión de memoria y el control de errores. C y C++ pueden dar como resultado aplicaciones muy potentes, pero debe asegurarse que el código funciona bien. Un error en la escritura del programa puede hacer que toda la aplicación falle o se comporte de forma inesperada. Como el objetivo al diseñar C++ era retener la compatibilidad con el anterior C, C++ fue incapaz de escapar de la naturaleza de bajo nivel de C.

Microsoft diseñó C# de modo que retuviera casi toda la sintaxis de C y C++. Los programadores que estén familiarizados con esos lenguajes pueden escoger el código C# y empezar a programar de forma relativamente rápida. Sin embargo, la gran ventaja de C# consiste en que sus diseñadores decidieron no hacerlo compatible con los anteriores C y C++. Aunque esto puede parecer un mal asunto, en realidad es una buena noticia. C# elimina las cosas que hacían que fuese difícil trabajar con C y C++. Como todo el código C es también código C++, C++ tenía que mantener todas las rarezas y deficiencias de C. C# parte de cero y sin ningún requisito de compatibilidad, así que puede mantener los puntos fuertes de sus predecesores y descartar las debilidades que complicaban las cosas a los programadores de C y C++.

- ***Introducción:***

C#, el nuevo lenguaje presentado en .NET Framework, procede de C++. Sin embargo, C# es un lenguaje orientado a objetos (desde el principio), moderno y seguro. Java, por ejemplo, también es un lenguaje de programación orientado a objetos, creado por Sun Microsystem a principios de los 90, que toma mucha de su sintaxis de C y C++, pero, del mismo modo que C#, tiene un modelo de objetos más simple y elimina herramientas de bajo nivel para evitar muchos errores, como la manipulación directa de punteros o memoria.

Por otro lado, Java tiene más años que C# y ha creado una gran base de actividad de usuarios convirtiéndose en la lengua franca en muchos sectores modernos de la informática, en particular, en las áreas de creación de redes. La gran mayoría de cursos en la educación secundaria y de nivel universitario de programación son dominados por Java, y actualmente hay más libros de Java que libros de C#. La pega, es que Java evoluciona muy lentamente y no ha incorporado a su lenguaje los nuevos patrones de programación y las nuevas metodologías.

- ***Características del lenguaje:***

Las siguientes secciones hacen un rápido repaso a algunas de las características de C#. Debido a la similitud de este lenguaje con Java, iremos comentando las diferencias y similitudes de ambos lenguajes en las siguientes características.

Clases:

Todo el código y los datos en C# y en Java deben ser incluidos en una clase. No puedo definir una variable fuera de una clase y no puede escribir ningún código que no esté en una clase. Las clases pueden tener **constructores**, que se ejecutan cuando se crea un objeto de la clase, de esta forma nos evitamos el tener que iniciar variables explícitamente para su iniciación. Y un destructor, que se ejecuta cuando un objeto de la clase es destruido. Las clases admiten herencias simples y todas las clases derivan al final de una clase base llamada objeto. C# admite técnicas de versiones para ayudar a que sus clases evolucionen con el tiempo mientras mantienen la compatibilidad con código que use versiones anteriores de sus clases.

El constructor tiene exactamente el mismo nombre de la clase que lo implementa; no puede haber ningún otro método que comparta su nombre con el de su clase. Una vez definido, se llamará automáticamente al constructor al crear un objeto de esa clase (al utilizar el operador new).

El constructor no devuelve ningún tipo, ni siquiera void. Su misión es iniciar todo estado interno de un objeto (sus atributos), haciendo que el objeto sea utilizable inmediatamente; reservando memoria para sus atributos, iniciando sus valores...

Por ejemplo, observe la clase llamada Family. Esta clase contiene los dos campos estáticos que incluyen nombre y el apellido de un miembro de la familia junto con un método que devuelve el nombre completo del miembro de la familia:

```
class Class1

{

        public string FistName;
        public string LastName;
        public string FullName;

        {
        return FirstName + LastName;
        }

}
```

El elemento básico de la programación orientada a objetos en C# y en Java es la clase. Una clase define la forma y el comportamiento de un objeto.

Para crear una clase sólo es necesario un archivo fuente que contenga la palabra clave reservada *class* seguida de un identificador legal y un bloque delimitado por dos llaves para el cuerpo de la clase, como se puede observar en el ejemplo anterior.

El archivo fuente, tanto en Java como en C#, debe tener el mismo nombre que el identificador de la clase que contiene, y en el caso de C# se les suele asignar la extensión ".cs", y en Java la extensión ".java". Por ejemplo, con el caso de ejemplo anterior, la clase Class1 se guardaría en un archivo Class1.cs o Class1.java.

C# permite trabajar con dos tipos de datos: de valor y de referencia. Los de valor contienen valores reales. Los de referencia contienen referencias a valores almacenados en algún lugar de la memoria. Los tipos primitivos como *char, int y float,* juntos con los valores y estructuras comentados, son tipos de valor. Los tipos de referencia tienen variables que tratan con objetos y matrices. C# viene con tipos de referencia predefinidos (*object y string*), junto con tipos de valor predefinidos (*sbyte, short, int, long,byte, ushort, uint, ulong, float, double, bool, char y decimal*). También puede definir en el código sus propios tipos de valor y referencia. Todos los tipos de valor y de referencia derivan en última instancia de un tipo base llamado *object.*

C# le permite convertir un valor de un tipo en un valor de otro tipo. Puede trabajar con conversiones implícitas y explícitas. Las conversiones implícitas siempre funcionan y nunca pierden información (por ejemplo, puede convertir un *int* en un *long* sin perder ningún dato porque un *long* es mayor que un *int*). Las conversiones explícitas pueden producir pérdidas de datos (por ejemplo, convertir un *long* en un *int* puede producir pérdida de datos porque un *long* puede contener valores mayores que un *int*). Debe escribir un operador *cast* en el código para que se produzca una conversión explícita.

En C# puede trabajar con matrices unidimensionales y multidimensionales. Las matrices multidimensionales pueden ser rectangulares, en las que cada una de las matrices tiene las mismas dimensiones, o escalonadas, en las que cada una de las matrices puede tener diferentes dimensiones.

Las clases y las estructuras pueden tener miembros de datos llamados *propiedades y campos*. Los campos son variables que están asociadas a la clase o estructura a la que pertenecen. Por ejemplo, puede definir una estructura llamada *Empleado* que tenga un campo llamado *Nombre.* Si define una variable de tipo *Empleado* llamada *EmpleadoActual,* puede recuperar el nombre del empleado escribiendo *EmpleadoActual.Nombre*. Las propiedades son como los campos, pero permiten escribir código que especifique lo que debería ocurrir cuando el código acceda al valor. Si el nombre del empleado debe leerse de una base de datos, por ejemplo, puede escribir código que diga "cuando alguien pregunte el valor de la propiedad *Nombre*, lee el nombre de la base de datos y devuelve el nombre como una cadena".

En Java, los atributos se pueden declarar con dos clases de tipos: un tipo simple Java (int, float, etc), o el nombre de una clase, que será una referencia a objeto. Cuando se realiza una instancia de una clase (creación de un objeto) se reservará en la memoria

un espacio para un conjunto de datos como el que definen los atributos de una clase. A este conjunto de variables se le denomina *variables de instancia.*

Funciones:

Una función es un fragmento de código que puede ser invocado y que puede o no devolver un valor al código que lo invoco en un principio. Un ejemplo de una función podría ser la función *FullName* mostrada anteriormente en este capítulo, en la clase *Family.* Una función suele asociarse a fragmentos de código que devuelven información, mientras que un método no suele devolver información. Sin embargo, para nuestros propósitos, generalizamos y nos referimos a las dos como funciones.

Las funciones pueden tener cuatro tipos de parámetros:

- Parámetros de entrada: tienen valores que son enviados a la función, pero la función no puede cambiar esos valores.
- Parámetros de salida: no tienen valor cuando son enviados a la función, pero la función puede darles un valor y enviar el valor de vuelta al invocarlos.
- Parámetros de referencia: introducen una referencia en otro valor. Tienen un valor de entrada para la función y ese valor puede ser cambiado dentro de la función.
- Parámetros Params: definen un número de variable de argumentos en una lista.

En Java, los métodos son subrutinas que definen la interfaz de una clase, sus capacidades y comportamiento. Un método ha de tener por nombre cualquier identificador legal distinto de los ya utilizados por los nombres de la clase en que está definido. Los métodos se declaran al mismo nivel que las variables de instancia dentro de una definición de clase. En Java, suele denominarse métodos en vez de funciones como en C#, devuelva el bloque de código un valor o no. Asi como en C#, en la declaración de los métodos (funciones en C#) se define el tipo de valor que devuelven y una lista formal de parámetros de entrada de sintaxis *tipo identificador* separadas por comas.

En el caso de que no se desee devolver ningún valor se deberá indicar como tipo la palabra reservada *void.* Asi mismo, si no se desean parámetros, la declaración del método debería incluir un par de paréntesis vacios.

```
void metodoVacio( ) { };
```

C# y el CLR trabajan juntos para brindar gestión de memoria automática. No necesita escribir código que diga “asigna suficiente memoria para un número entero” o “libera la memoria que está usando este objeto”. El CLR monitoriza el uso de memoria y recupera automáticamente más cuando la necesita. También libera memoria

automáticamente cuando detecta que ya no está siendo usada (esto también se conoce como recolección de objetos no utilizados).

En Java el problema de las fugas de memoria se evita en gran medida gracias a la recolección de basura (o automatic garbage collector). El programador determina cuándo se crean los objetos y el entorno en tiempo de ejecución de Java (Java runtime) es el responsable de gestionar el ciclo de vida de los objetos. El programa, u otros objetos pueden tener localizado un objeto mediante una referencia a éste. Cuando no quedan referencias a un objeto, el recolector de basura de Java borra el objeto, liberando así la memoria que ocupaba previniendo posibles fugas (ejemplo: un objeto creado y únicamente usado dentro de un método sólo tiene entidad dentro de éste; al salir del método el objeto es eliminado). Aun así, es posible que se produzcan fugas de memoria si el código almacena referencias a objetos que ya no son necesarios—es decir, pueden aún ocurrir, pero en un nivel conceptual superior. En definitiva, el recolector de basura de Java permite una fácil creación y eliminación de objetos, mayor seguridad y puede que más rápida que en C++

Java y C# proporcionan varios operadores que le permiten escribir expresiones matemáticas y de bits. Muchos (pero no todos) de estos operadores pueden ser redefinidos, permitiéndole cambiar la forma en que trabajan estos operadores.

C# como Java admiten una larga lista de expresiones que le permiten definir varias rutas de ejecución dentro del código. Las instrucciones de flujo de control que usan palabras clave como *if, switch, while, for, break y continue* permiten al código bifurcarse por caminos diferentes, dependiendo de los valores de sus variables. Las clases pueden contener código y datos. Cada miembro de una clase tiene algo llamado *ámbito de accesibilidad*, que define la visibilidad del miembro con respecto a otros objetos.

Variables:

Las variables pueden ser definidas como constantes. Las constantes tienen valores que no pueden cambiar durante la ejecución del código. Por ejemplo, el valor de pi es una buena muestra de una constante porque el valor no cambia a medida que el código se ejecuta. Las declaraciones de *tipo de enumeración* especifican un nombre de tipo para un grupo de constantes relacionadas. Por ejemplo, puede definir una enumeración de planetas con valores de Mercurio, Venus, Tierra, Marte, Júpiter, Saturno, Urano, Neptuno y Plutón, y usar estos nombres en el código. Usando los nombres de enumeraciones en el código hace que sea más fácil leerlo que si usara un número para representar a cada planeta.

C# incorpora un mecanismo para definir y procesar eventos. Si escribe una clase que realiza una operación muy larga, quizás quiera invocar a ese evento e incluirlo en el código, lo que permite que se les pueda avisar cuando haya acabado su operación. El

mecanismo de control de eventos en C# usa delegados, que son variables que se refieren a una función.

Si la clase contiene un conjunto de valores, los clientes quizás quieran acceder a los valores como si la clase fuera una matriz. Puede conseguirlo escribiendo un fragmento de código conocido como indexador. Suponga que escribe una clase llamada ArcoIris, por ejemplo, que contenga el conjunto de los colores del arco iris. Los visitantes querrán escribir MiArcoIrirs[0] para recuperar el primer color del arco iris. Puede escribir un indexador en la clase ArcoIris para definir lo que se debe devolver cuando el visitante acceda a esa clase, como si fuera una matriz de valores.

Interfaces:

C# admite interfaces, que son grupos de propiedades, métodos y eventos que especifican un conjunto de funcionalidad. Las clases C# pueden implementar interfaces, que informan a los usuarios de que la clase admite el conjunto de funcionalidades documentado por la interfaz. Puede desarrollar implementaciones de interfaces sin interferir con ningún código existente. Una vez que la interfaz ha sido publicada, no se puede modificar, pero puede evolucionar mediante herencia. Las clases C# pueden implementar muchas interfaces, aunque las clases solo pueden derivarse de una clase base.

Las interfaces Java son expresiones puras de diseño. Se trata de auténticas conceptualizaciones no implementadas que sirven de guía para definir un determinado concepto (clase) y lo que debe hacer, pero sin desarrollar un mecanismo de solución.

Se trata de declarar métodos abstractos y constantes que posteriormente puedan ser implementados de diferentes maneras según las necesidades de un programa.

Veamos un ejemplo de la vida real que puede beneficiarse del uso de interfaces para ilustrar su papel extremadamente positivo en C#. Muchas de las aplicaciones disponibles hoy en día admiten módulos complementarios. Supongamos que tiene un editor de código para escribir aplicaciones. Este editor, al ejecutarse, puede cargar módulos complementarios. Para ello, el modulo complementario debe seguir unas cuantas reglas. El modulo complementario de DLL debe exportar una función llamada CEEntry y el nombre de la DLL debe empezar por CEd. Cuando ejecutamos nuestro editor de código, este busca en su directorio de trabajo todas las DLL que empiecen por CEd. Cuando encuentra una, la abre y a continuación utiliza GetProcAddress para localizar la función CEEntry dentro de la DLL, verificando asi que ha seguido todas las reglas exigidas para crear un modulo complementario. Este método de creación y apertura de módulos complementarios es muy pesado porque sobrecarga el editor de código con más tareas de verificación de las necesarias. Si usáramos una interfaz en este caso, la DLL del módulo complementario podría haber implementado una

interfaz, garantizando así que todos los métodos necesarios, propiedades y eventos estén presentes en la propia DLL y que funciona como especifica la documentación.

Atributos:

Los atributos aportan información adicional sobre su clase al CLR. Antes, si quería que la clase fuera auto descriptiva, tenía que seguir un enfoque sin conexión, en el que la documentación fuera almacenada en archivos externos como un archivo IDL o incluso archivos HTML. Los atributos solucionan este problema permitiendo al programador vincular información a las clases (cualquier tipo de información). Por ejemplo, puede usar un atributo para insertar información de documentación en una clase, explicando cómo debe actuar al ser usada. Las posibilidades son infinitas y esa es la razón por la que Microsoft incluye tantos atributos predefinidos en .NET Framework.

- ***Comparativa entre ambos lenguajes:***

Desde los inicios de sus tiempos, siempre ha habido una fuerte discusión acerca de cuál de este par de lenguajes es mejor.

C# es un lenguaje de programación orientado a objetos desarrollado y estandarizado por Microsoft, mientras que Java es un lenguaje de programación orientado a objetos, desarrollado por Sun Microsystems, son las 2 caras de la moneda, aunque Java nació 6 años antes que C#, por allá en 1995, mientras que C# nació en el 2001, la sintaxis de los dos es bastante parecida, razón por la cual muchos dicen que C# es simplemente una copia de nuestro gran hermano Microsoft.

De cualquier manera los dos son muy buenos lenguajes de programación, aunque Java tiene una pequeña ventaja ya que es multiplataforma, mientras que C# es únicamente para Windows.

Hay unas muy buenas comparaciones en Internet. Me he encontrado 2 bastante buenas ya que con ejemplos concretos muestran las diferencias y similitudes de ambos lenguajes.

ESTRUCTURA DEL PROGRAMA	
Java	**C#**

package hello;
 	
public class HelloWorld {
  public static void **main**(String[] args) {
  String name = "Java";
 	
  // See if an argument was passed from the command line
  if (args.length == 1)
  name = args[0];
 	
  System.out.println("Hello, " + name + "!");
  }
}
	using System;
	
	namespace Hello {
	  public class HelloWorld {
	  public static void Main(string[] args) {
	  string name = "C#";
	
	  // See if an argument was passed from the command line
	  if (args.Length == 1)
	  name = args[0];
	
	  Console.WriteLine("Hello, " + name + "!");
	  }
	  }
	}

COMENTARIOS	
Java	**C#**
// Single line */* Multiple* * line */* */** Javadoc documentation comments */*	*// Single line* */* Multiple* * line */* */// XML comments on a single line* */** XML comments on multiple lines */*

CONSTANTES	
Java	**C#**

// May be initialized in a constructor
final double PI = 3.14;
	const double PI = 3.14;
	
	// Can be set to a const or a variable. May be initialized in a constructor.
	readonly int MAX_HEIGHT = 9;

TIPOS DE DATOS	
Java	**C#**

Primitive Types
boolean
byte
char
short, int, long
float, double
 	
Reference Types
Object *(superclass of all other classes)*
String
arrays, classes, interfaces
 	
Conversions
 	
// int to String
int x = 123;
String y = Integer.toString(x); *// y is "123"*
 	
// String to int
y = "456";
x = Integer.parseInt(y); *// x is 456*
 	
// double to int
double z = 3.5;
x = **(int)** z; *// x is 3 (truncates decimal)*
	Value Types
	bool
	byte, sbyte
	char
	short, ushort, int, uint, long, ulong
	float, double, decimal
	structures, enumerations
	
	Reference Types
	object *(superclass of all other classes)*
	string
	arrays, classes, interfaces, delegates
	
	Convertions
	
	// int to string
	int x = 123;
	String y = x.ToString(); *// y is "123"*
	
	// string to int
	y = "456";
	x = int.Parse(y); *// or x = Convert.ToInt32(y);*
	
	// double to int
	double z = 3.5;
	x = **(int)** z; *// x is 3 (truncates decimal)*

ENUMERACIONES	
Java	**C#**

enum Action {Start, Stop, Rewind, Forward};
 	
// Special type of class
enum Status {
Flunk(50), Pass(70), Excel(90);
private final int value;
Status(int value) { this.value = value; }
public int value() { return value; }
};
 	
Action a = Action.Stop;
if (a != Action.Start)
System.out.println(a); *// Prints "Stop"*
 	
Status s = Status.Pass;
System.out.println(s.value()); *// Prints "70"*
	enum Action {Start, Stop, Rewind, Forward};
	
	enum Status {Flunk = 50, Pass = 70, Excel = 90};
	
	No equivalent.
	
	Action a = Action.Stop;
	if (a != Action.Start)
	Console.WriteLine(a); *// Prints "Stop"*
	
	Status s = Status.Pass;
	Console.WriteLine((int) s); *// Prints "70"*

OPERADORES	
Java	**C#**
Comparison == < > <= >= != *Arithmetic* + - * / % *(mod)* / *(integer division if both operands are ints)* Math.Pow(x, y) *Assignment* = += -= *= /= %= &= |= ^= <<= >>= >>>= ++ -- *Bitwise* & | ^ ~ << >> >>> *Logical* && || & | ^ ! **Note:** && and || perform short-circuit logical evaluations *String Concatenation* +	*Comparison* == < > <= >= != *Arithmetic* + - * / % *(mod)* / *(integer division if both operands are ints)* Math.Pow(x, y) *Assignment* = += -= *= /= %= &= |= ^= <<= >>= ++ -- *Bitwise* & | ^ ~ << >> *Logical* && || & | ^ ! **Note:** && and || perform short-circuit logical evaluations *String Concatenation* +

CONDICIONALES/BIFURCACIONES	
Java	**C#**

greeting = age < 20 **?** "What's up?" **:** "Hello";
 	
if (x < y)
  System.out.println("greater");
 	
if (x != 100) {
  x *= 5;
  y *= 2;
}
else
  z *= 6;
 	
int selection = 2;
switch (selection) { *// Must be byte, short, int, char, or enum*
  case 1: x++; *// Falls through to next case if no break*
  case 2: y++; **break;**
  case 3: z++; **break;**
  default: other++;
}
	greeting = age < 20 **?** "What's up?" **:** "Hello";
	
	if (x < y)
	  Console.WriteLine("greater");
	
	if (x != 100) {
	  x *= 5;
	  y *= 2;
	}
	else
	  z *= 6;
	
	string color = "red";
	switch (color) { *// Can be any predefined type*
	  case "red": r++; **break;** *// break is mandatory; no fall-through*
	  case "blue": b++; **break;**
	  case "green": g++; **break;**
	  default: other++; **break;** *// break necessary on default*
	}

BUCLES	
Java	**C#**

while (i < 10)
  i++;
 	
for (i = 2; i <= 10; i += 2)
  System.out.println(i);
 	
do
  i++;
while (i < 10);
 	
for (int i **:** numArray) *// foreach construct*
  sum += i;
 	
// for loop can be used to iterate through any Collection
import java.util.ArrayList;
ArrayList<Object> list = new ArrayList<Object>();
list.add(10); *// boxing converts to instance of Integer*
list.add("Bisons");
list.add(2.3); *// boxing converts to instance of Double*
 	
for (Object o **:** list)
  System.out.println(o);
	while (i < 10)
	  i++;
	
	for (i = 2; i <= 10; i += 2)
	  Console.WriteLine(i);
	
	do
	  i++;
	while (i < 10);
	
	foreach (int i **in** numArray)
	  sum += i;
	
	// foreach can be used to iterate through any collection
	using System.Collections;
	ArrayList list = new ArrayList();
	list.Add(10);
	list.Add("Bisons");
	list.Add(2.3);
	
	foreach (Object o **in** list)
	  Console.WriteLine(o);

ARRAYS	
Java	**C#**

int nums**[]** = {1, 2, 3}; *or* int**[]** nums = {1, 2, 3};
for (int i = 0; i < nums.length; i++)
  System.out.println(nums[i]);
 	
String names[] = new String[5];
names[0] = "David";
 	
float twoD**[][]** = new float[rows][cols];
twoD[2][0] = 4.5;
 	
int**[][]** jagged = new int[5][];
jagged[0] = new int[5];
jagged[1] = new int[2];
jagged[2] = new int[3];
jagged[0][4] = 5;
	int[] nums = {1, 2, 3};
	for (int i = 0; i < nums.Length; i++)
	  Console.WriteLine(nums[i]);
	
	string[] names = new string[5];
	names[0] = "David";
	
	float[,] twoD = new float[rows, cols];
	twoD[2,0] = 4.5f;
	
	int[][] jagged = new int[3][] {
	  new int[5], new int[2], new int[3] };
	jagged[0][4] = 5;

FUNCIONES

Java

```
// Return single value
int Add(int x, int y) {
  return x + y;
}

int sum = Add(2, 3);
```

```
// Return no value
void PrintSum(int x, int y) {
  System.out.println(x + y);
}

PrintSum(2, 3);
```

```
// Primitive types and references are always passed by value
void TestFunc(int x, Point p) {
  x++;
  p.x++;      // Modifying property of the object
  p = null;   // Remove local reference to object
}

class Point {
  public int x, y;
}

Point p = new Point();
p.x = 2;
int a = 1;
TestFunc(a, p);
System.out.println(a + " " + p.x + " " + (p == null)
);  // 1 3 false
```

```
// Accept variable number of arguments
int Sum(int ... nums) {
  int sum = 0;
  for (int i : nums)
    sum += i;
  return sum;
}

int total = Sum(4, 3, 2, 1);   // returns 10
```

C#

```
// Return single value
int Add(int x, int y) {
  return x + y;
}

int sum = Add(2, 3);
```

```
// Return no value
void PrintSum(int x, int y) {
  Console.WriteLine(x + y);
}

PrintSum(2, 3);
```

```
// Pass by value (default), in/out-reference (ref), and out-reference (out)
void TestFunc(int x, ref int y, out int z, Point
p1, ref Point p2) {
  x++;  y++;  z = 5;
  p1.x++;      // Modifying property of the object
  p1 = null;   // Remove local reference to object
  p2 = null;   // Free the object
}

class Point {
  public int x, y;
}

Point p1 = new Point();
Point p2 = new Point();
p1.x = 2;
int a = 1, b = 1, c;   // Output param doesn't need initializing
TestFunc(a, ref b, out c, p1, ref p2);
Console.WriteLine("{0} {1} {2} {3} {4}",
  a, b, c, p1.x, p2 == null);   // 1 2 5 3 True

// Accept variable number of arguments
int Sum(params int[] nums) {
  int sum = 0;
  foreach (int i in nums)
    sum += i;
  return sum;
}

int total = Sum(4, 3, 2, 1);   // returns 10
```

STRINGS

Java

```
// String concatenation
String school = "Harding ";
school = school + "University";   // school is "Harding University"

// String comparison
String mascot = "Bisons";
if (mascot == "Bisons")    // Not the correct way to do
```

C#

```
// String concatenation
string school = "Harding ";
school = school + "University";   // school is "Harding University"

// String comparison
string mascot = "Bisons";
if (mascot == "Bisons")    // true
```

string comparisons
if (mascot.**equals**("Bisons")) *// true*
if (mascot.**equalsIgnoreCase**("BISONS")) *// true*
if (mascot.**compareTo**("Bisons") == 0) *// true*
 	
System.out.println(mascot.**substring**(2, 5)); *// Prints "son"*
 	
// My birthday: Oct 12, 1973
java.util.Calendar c = new java.util.GregorianCalendar(1973, 10, 12);
String s = String.format("My birthday: %1$tb %1$te, %1$tY", c);
 	
// Mutable string
StringBuffer buffer = new **StringBuffer**("two ");
buffer.**append**("three ");
buffer.**insert**(0, "one ");
buffer.**replace**(4, 7, "TWO");
System.out.println(buffer); *// Prints "one TWO three"*
	if (mascot.**Equals**("Bisons")) *// true*
	if (mascot.**ToUpper**().**Equals**("BISONS")) *// true*
	if (mascot.**CompareTo**("Bisons") == 0) *// true*
	
	Console.WriteLine(mascot.**Substring**(2, 3)); *// Prints "son"*
	
	// My birthday: Oct 12, 1973
	DateTime dt = new DateTime(1973, 10, 12);
	string s = "My birthday: " + dt.ToString("MMM dd, yyyy");
	
	// Mutable string
	System.Text.**StringBuilder** buffer = new System.Text.**StringBuilder**("two ");
	buffer.**Append**("three ");
	buffer.**Insert**(0, "one ");
	buffer.**Replace**("two", "TWO");
	Console.WriteLine(buffer); *// Prints "one TWO three"*

CAPTURA DE EXCEPCIONES	
Java	**C#**

// Must be in a method that is declared to throw this exception
Exception ex = new Exception("Something is really wrong.");
throw ex;
 	
try {
  y = 0;
  x = 10 / y;
} **catch** (Exception ex) {
  System.out.println(ex.getMessage());
} **finally** {
  // Code that always gets executed
}
	Exception up = new Exception("Something is really wrong.");
	throw up; *// ha ha*
	
	try {
	  y = 0;
	  x = 10 / y;
	} **catch** (Exception ex) { *// Variable "ex" is optional*
	  Console.WriteLine(ex.Message);
	} **finally** {
	  // Code that always gets executed
	}

NAMESPACES	
Java	**C#**

package harding.compsci.graphics;
 	
// Import single class
import harding.compsci.graphics.Rectangle;
 	
// Import all classes
import harding.compsci.graphics.*;
	namespace Harding.Compsci.Graphics {
	  ...
	}
	
	or
	
	namespace Harding {
	  namespace Compsci {
	  namespace Graphics {
	  ...
	  }
	  }
	}
	
	// Import single class
	using Rectangle =
	Harding.CompSci.Graphics.Rectangle;
	
	// Import all class
	using Harding.Compsci.Graphics;

CLASES/INTERFACES	
Java	**C#**

Accessibility keywords
public
private
protected
static
 	
// Inheritance
class FootballGame **extends** Competition {
  ...
}
 	
// Interface definition
interface IAlarmClock {
  ...
}
 	
// Extending an interface
interface IAlarmClock **extends** IClock {
  ...
}
	Accessibility keywords
	public
	private
	internal
	protected
	protected internal
	static
	
	// Inheritance
	class FootballGame **:** Competition {
	  ...
	}
	
	// Interface definition
	interface IAlarmClock {
	  ...
	}
	
	// Extending an interface
	interface IAlarmClock : IClock {
	  ...
	}

```
// Interface implementation
class WristWatch implements IAlarmClock, ITimer {
   ...
}
```

```
// Interface implementation
class WristWatch : IAlarmClock, ITimer {
   ...
}
```

CONSTRUCTORES/DESTRUCTORES	
Java	**C#**

class SuperHero {
  private int mPowerLevel;
 	
  public SuperHero() {
  mPowerLevel = 0;
  }
 	
  public SuperHero(int powerLevel) {
  this.mPowerLevel= powerLevel;
  }
 	
  // No destructors, just override the finalize method
  protected void **finalize**() throws Throwable {
  super.finalize(); // Always call parent's finalizer
  }
}
	class SuperHero {
	  private int mPowerLevel;
	
	  public SuperHero() {
	  mPowerLevel = 0;
	  }
	
	  public SuperHero(int powerLevel) {
	  this.mPowerLevel= powerLevel;
	  }
	
	  ~SuperHero() {
	  // Destructor code to free unmanaged resources.
	  // Implicitly creates a Finalize method.
	  }
	}

OBJETOS	
Java	**C#**

SuperHero hero = new SuperHero();
 	
hero.setName("SpamMan");
hero.setPowerLevel(3);
 	
hero.Defend("Laura Jones");
SuperHero.Rest(); // Calling static method
 	
SuperHero hero2 = hero; // Both refer to same object
hero2.setName("WormWoman");
System.out.println(hero.getName()); // Prints WormWoman
 	
hero = **null**; // Free the object
 	
if (hero == **null**)
  hero = new SuperHero();
 	
Object obj = new SuperHero();
System.out.println("object's type: " +
obj.**getClass()**.toString());
if (obj **instanceof** SuperHero)
  System.out.println("Is a SuperHero object.");
	SuperHero hero = new SuperHero();
	
	hero.Name = "SpamMan";
	hero.PowerLevel = 3;
	
	hero.Defend("Laura Jones");
	SuperHero.Rest(); // Calling static method
	
	SuperHero hero2 = hero; // Both refer to same object
	hero2.Name = "WormWoman";
	Console.WriteLine(hero.Name); // Prints WormWoman
	
	hero = **null** ; // Free the object
	
	if (hero == **null**)
	  hero = new SuperHero();
	
	Object obj = new SuperHero();
	Console.WriteLine("object's type: " +
	obj.**GetType()**.ToString());
	if (obj **is** SuperHero)
	  Console.WriteLine("Is a SuperHero object.");

PROPIEDADES	
Java	**C#**

private int mSize;
 	
public int **getSize**() { return mSize; }
public void **setSize**(int value) {
  if (value < 0)
  mSize = 0;
  else
  mSize = value;
}
 	
int s = shoe.getSize();
shoe.setSize(s+1);
	private int mSize;
	
	public int Size {
	  get { return mSize; }
	  set {
	  if (value < 0)
	  mSize = 0;
	  else
	  mSize = value;
	  }
	}
	
	shoe.Size++;

ESTRUCTURAS	
Java	**C#**

No structs in Java.	**struct** StudentRecord {
	  public string name;
	  public float gpa;
	
	  public StudentRecord(string name, float gpa) {
	  this.name = name;
	  this.gpa = gpa;
	  }
	}
	
	StudentRecord stu = new StudentRecord("Bob", 3.5f);
	StudentRecord stu2 = stu;
	
	stu2.name = "Sue";
	Console.WriteLine(stu.name); *// Prints "Bob"*
	Console.WriteLine(stu2.name); *// Prints "Sue"*

CONSOLE I/O	
Java	**C#**

java.io.DataInput in = new
java.io.DataInputStream(System.in);
System.out.print("What is your name? ");
String name = in.readLine();
System.out.print("How old are you? ");
int age = Integer.parseInt(in.readLine());
System.out.println(name + " is " + age + " years old.");
 	
int c = System.in.read(); *// Read single char*
System.out.println(c); *// Prints 65 if user enters "A"*
	Console.Write("What's your name? ");
	string name = Console.ReadLine();
	Console.Write("How old are you? ");
	int age = Convert.ToInt32(Console.ReadLine());
	Console.WriteLine("{0} is {1} years old.", name, age);
	// or
	Console.WriteLine(name + " is " + age + " years old.");
	
	int c = Console.Read(); *// Read single char*
	Console.WriteLine(c); *// Prints 65 if user enters "A"*

// The studio costs $499.00 for 3 months. System.out.printf("The %s costs $%.2f for %d months.%n", "studio", 499.0, 3); *// Today is 06/25/04* System.out.printf("Today is %tD\n", new java.util.Date());	*// The studio costs $499.00 for 3 months.* Console.WriteLine("The {0} costs {1:C} for {2} months.\n", "studio", 499.0, 3); *// Today is 06/25/2004* Console.WriteLine("Today is " + DateTime.Now.ToShortDateString()); (stu2.name); *// Prints "Sue"*

FILE I/O	
Java	**C#**

import java.io.*;
 	
// Character stream writing
FileWriter writer = new FileWriter("c:\\myfile.txt");
writer.write("Out to file.\n");
writer.close();
 	
// Character stream reading
FileReader reader = new FileReader("c:\\myfile.txt");
BufferedReader br = new BufferedReader(reader);
String line = br.readLine();
while (line != null) {
  System.out.println(line);
  line = br.readLine();
}
reader.close();
 	
// Binary stream writing
FileOutputStream out = new FileOutputStream("c:\\myfile.dat");
out.write("Text data".getBytes());
out.write(123);
out.close();
 	
// Binary stream reading
FileInputStream in = new FileInputStream("c:\\myfile.dat");
byte buff[] = new byte[9];
in.read(buff, 0, 9); *// Read first 9 bytes into buff*
String s = new String(buff);
int num = in.read(); *// Next is 123*
in.close();
	using System.IO;
	
	// Character stream writing
	StreamWriter writer = File.CreateText("c:\\myfile.txt");
	writer.WriteLine("Out to file.");
	writer.Close();
	
	// Character stream reading
	StreamReader reader = File.OpenText("c:\\myfile.txt");
	string line = reader.ReadLine();
	while (line != null) {
	  Console.WriteLine(line);
	  line = reader.ReadLine();
	}
	reader.Close();
	
	// Binary stream writing
	BinaryWriter out = new BinaryWriter(File.OpenWrite("c:\\myfile.dat"));
	out.Write("Text data");
	out.Write(123);
	out.Close();
	
	// Binary stream reading
	BinaryReader in = new BinaryReader(File.OpenRead("c:\\myfile.dat"));
	string s = in.ReadString();
	int num = in.ReadInt32();
	in.Close();

2.2.Objetivos:

El objetivo de este proyecto es la creación de una aplicación para el entretenimiento utilizando las ventajas ofrecidas por Unity 3D para este tipo de aplicaciones. Unity 3D es un programa para realizar juegos para internet, Play Station 3 y lo más importante para móviles Iphone, móviles con sistema Android y cualquiera que soporte Unity. En este proyecto la programación se realizará con el lenguaje C#.

Los objetivos a desarrollar, en una primera instancia, serían los siguientes:

- Estudio de las herramientas, lenguajes y programas a utilizar en el diseño de la aplicación
- Diseño de la aplicación.
- Implementación de la aplicación.
- Pruebas de la aplicación.
- Documentación de la aplicación para el correcto manejo del interface.

2.3. Fases del proyecto:

Etapas de desarrollo

- En primer lugar, realizaremos una etapa de formación para estudiar el lenguaje C# para su posterior empleo en la aplicación. Realizaremos varios tutoriales para el estudio del programa Unity 3d donde realizaremos la implementación de la aplicación.
- En segundo lugar, es necesario realizar cuatro iteraciones (prototipos) para ir mejorando de manera progresiva nuestra aplicación.

Cada iteración constará de las siguientes fases:

- **Análisis**: Analizaremos los requisitos y requerimientos.
- **Diseño del sistema**: Se empezará por lo más general y se irá bajando hasta llegar al nivel más bajo de diseño. Seguiremos una metodología de Diseño Centrado en el Usuario.
- **Implementación**: A través de la herramienta Unity 3D en lenguaje C#.
- **Pruebas:** Periodo en el que se realizarán pruebas a nivel general del sistema, corrigiendo los errores y fallos que se vayan presentando.
- **Documentación:** Se harán informes en donde se muestre el progreso al terminar cada iteración del proyecto así como la elaboración de la documentación necesaria para el uso de la aplicación creada. Además de la entrega final (memoria del proyecto) con todos los datos.

Por último, realizaremos los manuales necesarios para la perfecta utilización de la aplicación.

2.4. Planificación:

PLANIFICACIÓN			
Nombre Etapa	**Fecha comienzo**	**Fecha fin**	**Duración**
Formación	21/10/2010	21/11/2010	1 mes
Iteración 1	21/11/2010	21/01/2011	2 meses
Iteración 2	21/01/2011	21/02/2011	1 mes
Iteración 3	21/02/2011	21/03/2011	1 mes
Iteración 4	21/03/2011	21/04/2011	1 mes
Documentación y memoria	21/04/2011	21/05/2011	1 mes

3. Desarrollo:

3.1.Estudio del software social:

Cuando decidimos realizar un juego para un iPad, nos fuimos guiando por los juegos y aplicaciones que ya existían para hacernos a la idea de cómo es un juego para iPad y cuáles son las características que debe cumplir.

Observamos que la mayoría de juegos para iPad se aprovechaban de la característica del acelerómetro que tiene el dispositivo, con la cual se controla la orientación del

dispositivo y como se puede observar en la imagen, en ese juego, en concreto, se utilizaba para controlar la dirección del coche del famoso juego Need For Speed.

La gran mayoría de juegos de la red para iPad eran juegos simples, de realizar una función, sin mucha calidad gráfica, si no juegos intuitivos para facilitar la usabilidad, como en el ejemplo de esta imagen, donde tienes un tiempo delimitado y como objetivo realizar una parábola con la pelota de baloncesto para conseguir canastas.

La gran mayoría de los juegos consistía en lanzar un objeto, con más o menos fuerza, con una dirección o con otra, para cumplir un objetivo, como en el caso del juego Let´s golf.

Y también, muy importante el aprovechamiento de las características al máximo del dispositivo sobre el que va a correr el juego, como en el siguiente juego, “Boomerang AR”, en el cual se lanza un boomerang moviendo el dispositivo.

3.2.Formación en Unity 3D:

En los apartados anteriores ya hemos explicado el funcionamiento básico del programa Unity 3D, y a continuación, vamos a explicar las clases más comunes que se utilizan para construir una aplicación en Unity.

Las clases son las siguientes:

1. Collision
2. ContactPoint
3. Debug
4. GUI
5. Input
6. Object
    a. Component
        i. Behaviour
            1. AudioSource
            2. AudioListener
            3. GUIElement
                a. GUIText
                b. GUITexture
            4. Light
            5. Monobehaviour
        ii. Collider
        iii. ParticleAnimator
        iv. ParticleEmitter
        v. Rigidbody
        vi. Transform
    b. GameObject
    c. Texture
7. Ray
8. RaycastHit
9. Screen
10. Time
11. Touch
12. Vector3

Después de explicar las clases, comentaremos el proceso de compilación y ejecución de la aplicación en un iPad, es decir, el proceso de pasar la aplicación de Unity al propio iPad

- **Clases de Unity:**

1) Collision:

Describe las colisiones de los objetos. La información de las colisiones es pasada por los eventos:

- Collider.OnCollisionEnter:

  Este evento es llamado cuando el collider/rigidbody ha comenzado a tocar a otro rigidbody/collider. A diferencia de OnTriggerEnter, en OnCollisionEnter es pasado como parámetro el Collision class y no un Collider. La Collision class contiene información sobre puntos de contacto (contact points), velocidad de impacto (impact velocity), etc.

- Collider.OnCollisionStay:

  Es llamado una vez por frame para todo collider/rigidbody que está tocando otro rigidbody/collider. A diferencia de OnTriggerStay, OnCollisionStay pasa la clase Collision y no un Collider.

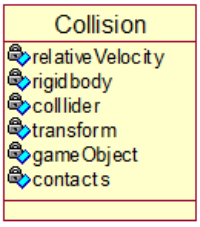
- Collider.OnColliderExit:

  Este evento, es llamado cuando el collider/rigidbody ha dejado de tocar otro rigidbody/collider. Como en los anteriores, este evento también pasa la clase Collision en vez de un Collider.

La clase **Collision** tiene unas cuantas variables que contienen información sobre la colisión:

relativeVelocity	La velocidad lineal relativa de los dos objetos de la colisión (solo lectura)
rigidbody	El Rigidbody que golpeamos, esta variable será nula si el collider golpeado no contiene el componente rigidbody
collider	El Collider golpeado
transform	El componente Transform que es golpeado
gameObject	/gameObject/ es el objeto con el que estamos chocando
contacts	Los puntos de contacto generados por la física de los objetos.

Veamos ahora el diagrama de la clase Collision:



2) ContactPoint:

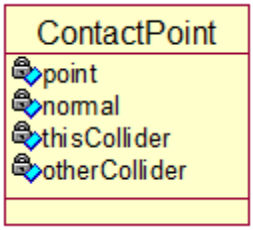
Describe el **punto** de contacto cuando sucede una **colisión**.

Los puntos de contacto son almacenados en la estructura **Collision**.

Las **variables** de información son:

point	El punto de contacto
normal	La normal de el punto de contacto
thisCollider	El primer collider en contacto
otherCollider	El otro collider en contacto

Veamos el **diagrama de la clase**:



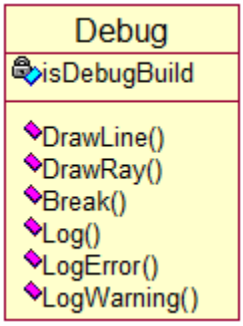
3) Debug:

Es la clase que contiene los métodos para realizar un **debug** de la manera más sencilla desarrollando un juego.

Las **funciones** de la clase son:

DrawLine	Dibuja una línea en el punto en el cual comienza y acaba con color
DrawRay	Dibuja una línea de start to start + dir con color
Break	Pausa el editor
Log	Mandas un mensaje a la consola de Unity
LogError	Es una variante de Debug.Log que manda un mensaje de error a la consola.
LogWarning	Es una variante también, que manda un mensaje de aviso a la consola de Unity.

Veamos el **diagrama de la clase:**



4) GUI:

La clase GUI es el interfaz para los GUI de Unity con posicionamiento manual.

Su **constructor** es GUI (static function GUI () : GUI)

Las **variables** de la clase son:

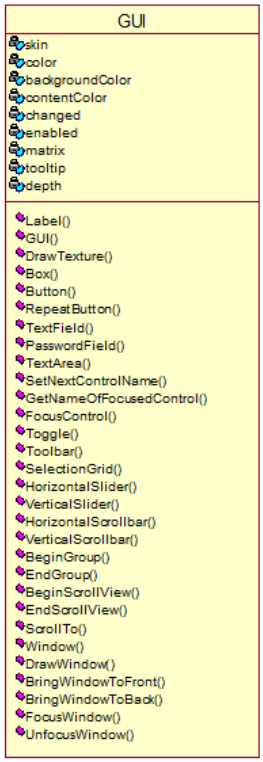
skin	El skin global para usar
color	Tinte de color global para la GUI
backgroundColor	Color global de fondo para todos los elementos de laGUI
contentColor	Color para todos los textos renderizados por GUI
changed	Devuelve **true** si algún control cambia de valor
enabled	Está el GUI habilitado?
matrix	El GUI transforma matrix
tooltip	Una descripción de la posición del ratón sobre los elementos GUI

Las **funciones** de la clase:

Label	Crea un texto o un texture label en la pantalla
DrawTexture	Dibuja una textura dentro de un rectángulo
Box	Crea un graphical box
Button	Crea un simple press button.
RepeatButton	Crea un botón que se activa siempre que el usuario mantenga pulsado.
TextField	Crea un text field donde el usuario puede modificar el texto
PasswordField	Crea un text field donde el usuario puede introducir la contraseña
TextArea	Crea un multi-line text área, donde el usuario puede editar texto
SetNextControlName	Pone el nombre para el próximo control
GetNameOfFocusedControl	Adquiere el nombre del control llamado.
FocusControl	Mueve el foco de teclado para el control nombrado

Toggle	Crea un on/off toggle button
Toolbar	Crea una barra de herramientas
SelectionGrid	Crea una red de botones
HorizontalScrollbar	Crea una barra de desplazamiento horizontal
VerticalScrollbar	Crea una barra de desplazamiento vertical
BeginGroup	Comienza un grupo, y debe ir acompañada con una llamada a EndGroup
EndGroup	Finaliza un grupo
BeginScrollView	Comienza un scrolling view dentro de GUI
EndScrollView	Termina un scrolling view
ScrollTo	Desplaza todos los scrollviews adjuntando por lo que tratar de hacer lugar visible
Window	Crea un popup window
DragWindow	Hace una ventana movible
BringWindowToFront	Traiga una ventana específica al frente de las ventanas flotantes.
BringWindowToBack	Lleve una ventana especifica al fondo de las ventanas
FocusWindow	Hacer que una ventana se convierta en la ventana activa
UnfocusWindow	Eliminar el foco sobre todas las ventanas

Veamos el **diagrama de clase** de GUI donde podremos observar sus variables (atributos) y funciones:



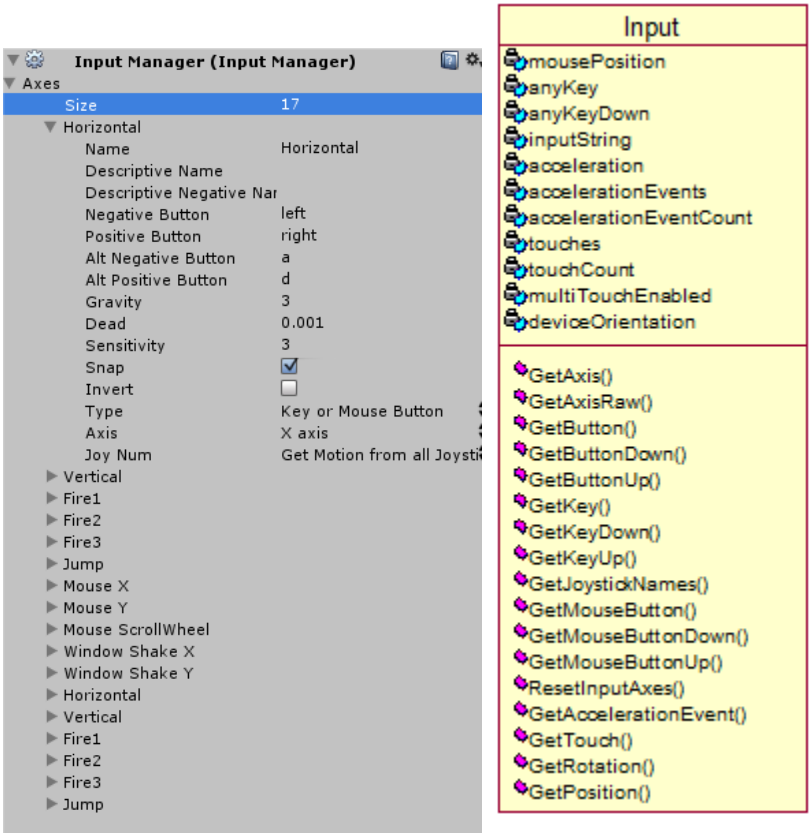
5) Input:

Interfaz en el input system (sistema de entrada).

Esta clase es usada para leer los ejes establecidos en el Input Manager, y para acceder a los datos del multi-touch/accelerometro de los dispositivos móviles.

Para leer un eje se usa **Input.GetAxis** con uno de los siguientes ejes por defecto: “Horizontal” y “vertical” son asignados en el joystick, A, W, S, D y las flechas del teclado. “Mouse X” y “Mouse Y” son asignados en el ratón. “Fire1”, “Fire2”, “Fire3” son asignados para las teclas Ctrl, Alt, Cmd y tres botones del raton o joystick. Nuevos ejes de entrada (input) pueden ser añadidos en el Input Manager.

A continuación, veremos dos imágenes, una en la que se muestra el interfaz del **InputManager** y otra en la cual se verá el diagrama de la clase.



Como se puede observar en la anterior imagen, en el **InputManager** se encuentran todas las funciones asignadas, 17 en total, pero que pueden añadirse más si modificamos el valor de la variable “size” para asignar una nueva función si lo considerásemos oportuno.

Si tú estás usando input para cualquier modo de comportamiento de un movimiento usa **Input.GetAxis.** Ésta te proporcionará una entrada configurable que puede ser asignada al teclado, joystick o el ratón. Usa **Input.GetButton** para acciones como

eventos y no lo uses para movimientos, ya que **Input.GetAxis** hará el código del script más pequeño y simple.

Input en dispositivos móviles:

Los dispositivos iOS y Android están capacitados para captar múltiples toques de dedo en la pantalla simultáneamente. Puedes acceder a los datos del estado de ese toque de dedo (finger touching screen) en el anterior frame accediendo al array de la propiedad **Input.touches**.

Cuando un dispositivo se mueve, su hardware acelerómetro envía una señal líneal de aceleración que cambia en los tres ejes primarios de las tres dimensiones del espacio. Puedes utilizar esa información para detectar la orientación actual del dispositivo y si esta orientación cambia.

El hardware de aceleración envía los valores de orientación de los ejes del dispositivo en forma de fuerzas G, de tal forma que un valor de 1.0 representa una fuerza de +1G en ese eje, mientras que un valor de -1.0 representa una fuerza de -1G.

Para obtener los datos del acelerómetro puedes leer la propiedad **Input.acceleration**, aunque también puedes utilizar la propiedad **Input.deviceOrientation** para saber exactamente la orientación del dispositivo en el eje que quieras.

Las **variables** de la clase son:

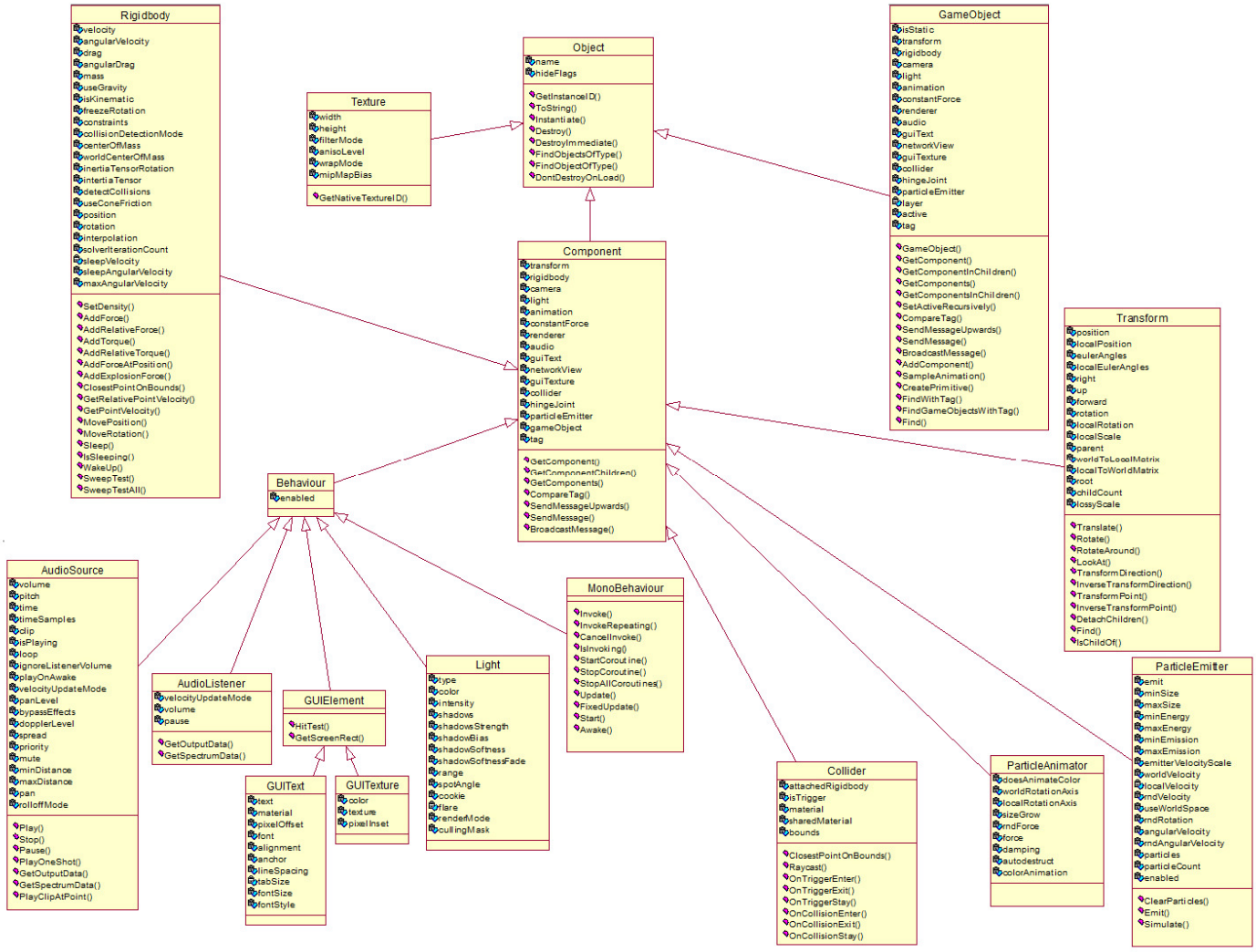
mousePosition	La posición del ratón en coordenadas pixel.
anyKey	Está alguna tecla o botón del ratón siendo pulsada?
anyKeyDown	Devuelve true en el primer frame que el usuario presione cualquier botón.
inputString	Devuelve la input de teclado entrada en este frame
acceleration	Última aceleración del dispositivo en los tres ejes
accelerationEvents	La lista de compases de aceleración en el anterior frame
accelerationEventCount	Número de aceleraciones en el anterior frame
touches	Devuelve la lista de objetos que representan el estado de todos los touches en el último frame.
touchCount	El número de touches.
multiTouchEnabled	Indica si el sistema se encarga de multiTouch
deviceOrientation	Orientación del dispositiva física

Las **funciones** de la clase son:

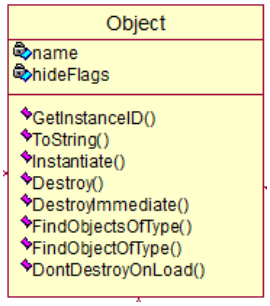
GetAxis	Devuelve el valor del eje virtual identificado por axisName
GetAxisRaw	Devuelve el valor del eje virtual identificado por axisName, sin aplicar filtro smoothing.
GetButton	Devuelve true mientras el botón virtual este pulsado
GetButtonDown	Devuelve true durante el frame que el botón se encuentra pulsado
GetButtonUp	Devuelve true en el primer frame que el usuario deja de pulsar el botón.
GetKey	Devuelve true mientras el usuario presiona la tecla identificada por el nombre
GetKeyDown	Devuelve true durante el frame que el usuario comienza a presionar la tecla
GetKeyUp	Devuelve true durante el frame que el usuario deja de presionar la tecla
GetJoystickNames	Devuelve un array con la descripción de los joystick conectados
GetMouseButton	Indica si el botón está siendo pulsado
GetMouseButtonDown	Devuelve true durante el frame que el usuario presiona el botón.
GetMouseButtonUp	Devuelve true durante el frame que el usuario deja de presionar el botón.
ResetInputAxes	Resetea todas las entradas (inputs)
GetAccelerationEvent	Devuelve la descripción de la aceleración del ultimo frame
GetTouch	Devuelve el objeto que representa el estado del touch
GetRotation	
GetPosition	

6) Object:

Es la clase base de todos los objetos que Unity puede hacer referencia. Cualquier variable que derive de un objeto podrá verse en el inspector permitiéndote dar un valor en el interfaz, sin tener que modificar el script.



En esta imagen vemos el **diagrama de clases** de la clase **Object**, en el cual vemos las relaciones de herencia con las clases más importantes y utilizadas en nuestro proyecto, como son **Rigidbody, Transform, GameObject, Texture, Component**, y muchas más.



Las **variables** de la clase son:

name	El nombre del objeto
hideFlags	Si el objeto se oculta, salva la escena o modifica por el usuario.

Las **funciones**:

GetInstanceID	Devuelve el identificador de la instancia del objeto
ToString	Devuelve el nombre de el objeto

Las **funciones** de la clase (class functions):

Operador bool	Existe el objeto?
Instantiate	Clona el objeto original y devuelve el clone
Instantiate.<T>	
Destroy	Elimina un gameobject, componente o asset
DestroyImmediate	Destruye el objeto inmediatamente.
FindObjectsOfType	Devuelve una lista de objetos cargados del mismo tipo
FindObjectOfType	Devuelve el primer objeto activo del mismo tipo
Operador ==	Compara si dos objetos hacen referencia al mismo
Operador!=	Compara si dos objetos no hacen referencia al mismo.
DontDestroyOnLoad	Hace que el objeto de destino no se destruya automáticamente cuando se carga una nueva escena.

AnimationClip, AssetBundle AudioClip **Component**, Flare, Font, **GameObject**, Material, Mesh, PhysicMaterial, ScriptableObject, Shader, TerrainData, TextAsset y **Texture**, estas son las clases que heredan de la clase **Object**, por lo que solo explicaremos las clases que hemos utilizado en la creación de nuestra aplicación.

a) Component(hereda de Object):

Es la clase base para todos los **componentes** que van unidos a un gameObject. Esta clase hereda todos los miembros de la clase **Object** anteriormente nombrados.

Las **variables** son:

transform	El Transform que está unido al GameObject (null si no hay unido)
rigidbody	El Rigidbody que está unido al GameObject (null si no hay unido)
camera	El Camera que está unido al GameObject (null si no hay unido)
light	El Light que está unido al GameObject (null si no hay unido)
animation	El Animation que está unido al GameObject (null si no hay unido)
constantForce	El ConstantForce que está unido al GameObject (null si no hay unido)

renderer	El Renderer que está unido al GameObject (null si no hay unido)
audio	El AudioSource que está unido al GameObject (null si no hay unido)
guiText	El GUIText que está unido al GameObject (null si no hay unido)
networkView	El NetworkView que está unido al GameObject (null si no hay unido)
guiTexture	El GUITexture que está unido al GameObject (null si no hay unido)
collider	El Collider que está unido al GameObject (null si no hay unido)
hingeJoint	El HingeJoint que está unido al GameObject (null si no hay unido)
particleEmitter	El ParticleEmitter que está unido al GameObject (null si no hay unido)
gameObject	El game object al que están unidos los componentes
tag	El tag del game object.

Las **funciones** son:

GetComponent	Devuelve el componente del tipo o del nombre pasado por parámetro si el objeto tiene unido alguno, si no, devuelve null
GetComponentInChildren	Devuelve el componente del tipo seleccionado del gameObject o de alguno de sus hijos usando la búsqueda en profundidad
GetComponentsInChildren	Devuelve los componentes del tipo seleccionado del GameObject o alguno de sus hijos
GetComponents	Devuelve todos los componentes del tipo seleccionado que tiene el GameObject
CompareTag	Este GameObject está etiquetado como tag?
SendMessageUpwards	Invoca al método llamado methodName en cualquier script de ese gameObject y en todos los ancestros del gameObject
SendMessage	Invoca al método llamado methodName en cualquier script de ese gameObject
BroadcastMessage	Invoca al método llamado methodName en cualquier script de ese gameObject y todos sus hijos.

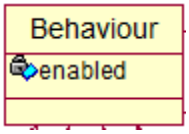
En Unity hay varias clases que **heredan** de la clase base **Component**. Estas son Behaviour, Cloth, Collider, Joint, MeshFilter, OcclusionArea, ParticleAnimator, ParticleEmitter, Renderer, Rigidbody, TextMesh, Transform y Tree.

i) Behaviour (hereda de Component):

Behaviour son componentes que se pueden habilitar o deshabilitar. Solo contienen una variable, **enabled**, y con ella habilitan o deshabilitan los componentes.

Poseen muchos miembros heredados tanto de **components** como de **object.**

La siguiente imagen corresponde con el **diagrama** de la clase Behaviour.



(1) AudioSource (hereda de Behaviour):

A un **GameObject** se le une un componente **AudioSource** para reproducir música de fondo con tecnología 3D. Para reproducir sonidos 3D necesitas tener un **AudioListener**. El **AudioListener** normalmente está unido en la camera que el usuario utiliza. Los sonidos estéreo son reproducidos siempre sin atenuación por la distancia.

Puede reproducir un único clip de audio usando **Play, Pause,** y **Stop.** También, puedes ajustar el volumen del sonido con la propiedad **volumen**. Se pueden reproducir múltiples sonidos en uno utilizando **PlayOneShot.** Puedes reproducir sonidos en una posición estática en el espacio 3D usando **PlayClipAtPoint.**

Las **variables** de la clase son:

volume	El volumen del audio source
pitch	El pitch del audio source
time	Reproducción de la posición en segundos
timeSamples	Reproducción de la posición en PCM samples.
clip	El AudioClip por defecto para reproducir
isPlaying	El clip se está reproduciendo correctamente?
loop	El clip de audio está en un bucle?
ignoreListenerVolime	Hace que audio source no tenga en cuenta el volumen del audio listener
playOnAwake	Si le asignamos true, el audio source automáticamente empezará a reproducirse dentro de awake
velocityUpdateMode	Si audio source debe ser actualizada en fijos o dinámicos
panLevel	Determina cuánta tecnología 3D va a tener efecto en el canal
bypassEffects	El efecto bypassEffect
dopplerLevel	Indica la escala doppler del AudioSource
spread	Establece el ángulo de propagación del sonido.
priority	Establece la prioridad del AudioSource

mute	Silencia o no el audio. Para silenciar mute=0
minDistance	La mínima distancia a la que dejará de aumentar el volumen
maxDistance	La distancia máxima a la que el sonido dejará de atenuarse
pan	Establece canales pan de forma lineal. Sólo funciona en 2D clip
rolloffMode	Establece o obtiene el modo en el que AudioSource se atenúa fuera de distancia

Las **funciones** de la clase son:

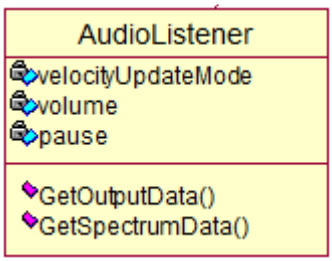
Play	Reproduce el clip
Stop	Para de reproducir el clip
Pause	Pausa la reproducción
PlayOneShot	Reproduce un AudioClip
GetOutputData	Devuelve un bloque de datos output de audio que se están reproduciendo
GetSpectrumData	Devuelve un bloque de datos spectrum de audio que se están reproduciendo

(2) AudioListener (hereda de Behaviour):

Representa a un **listener** en el espacio 3D. Este componente graba todos los sonidos que se reproducen a su alrededor y los reproduce para que el usuario los escuche. Solo se puede tener un **listener** en cada escena.

Posee una variable **velocityUpdateMode** con la que el audio **listener** puede ser actualizado en fijo o dinámico. Con **volumen** puedes controlar el volumen del juego y con **pause** puedes parar el estado del audio. Si pause es true el audio no se oirá.

El **diagrama** es el siguiente:



(3) GUIElement (hereda de Behaviour):

Es la clase básica para imágenes y texto mostrados en la interfaz de usuario (GUI). Esta clase contiene todas las funcionalidades de cualquier elemento GUI.

Las **funciones** son:

HitTest	Es un punto en la pantalla dentro del elemento
GetScreenRect	Devuelve un rectángulo que delimita el GUIElement en cordenadas de pantalla

Esta de esta clase hereda dos clases importantes:

(a) GUIText (hereda de GUIElement):

Es una frase de texto que se muestra en un interfaz (GUI).

Las **variables** son:

text	El texto para mostrar
material	El material a usar en el renderizado
pixelOffset	Los pixel de desplazamiento del texto
font	La tipografía de letra para el texto
alignment	La alineación del texto
anchor	El anclaje del texto
lineSpacing	El espacio interlineal
tabSize	El tamaño del tabulador
fontSize	El tamaño de fuente
fontStyle	El estilo de fuente (cursiva, negrita, …)

(b) GUITexture (hereda de GUIElement):

Es una textura usada en 2D GUI mediante una imagen.

Contiene tres **variables:**

color	El color de la GUI texture
texture	La textura a dibujar en pantalla
pixelInset	Es usada para ajustar por tamaño y posición

(4) Light (hereda de Behaviour):

Esta clase se usa para controlar el aspecto de las **luces** en Unity. Las propiedades se muestran también en el inspector, y lo más usual es cambiar y controlar los valores de la luz mediante el inspector, aunque otras veces puede ser de utilidad modificar estos valores en código.

Las **variables** son:

type	El tipo de luz
color	El color de la luz
intensity	La intensidad que es multiplicada por el color de la luz
shadows	Cómo quieres que muestre las sombras?
shadowStrength	La fuerza de las sombras de la luz
shadowBias	Sombra en Bias
shadowSoftness	La suavidad de las sombras de las luces direccionales
shadowSoftnessFade	La velocidad de suavidad de las sombras
range	El rango al que afecta la luz
spotAngle	Los ángulos de inclinación de la luz
cookie	La textura cookie proyectada por la luz
flare	El flare asset a usar para esta luz
renderMode	El modo de renderizar la luz
cullingMask	Es usado para iluminar partes de la escena selectivamente.

(5) MonoBehaviour (hereda de Behaviour):

Es la clase de la que derivan o heredan todos los script de la aplicación.

Cuando usas Javascript todos los scripts automáticamente derivan de **MonoBehaviour**. Sin embargo, cuando utilizamos C# o Boo, tenemos que especificar que hereda o deriva de MonoBehaviour.

La única **variable** que maneja esta clase es **useGUILayout,** que se usa para saltar la fase de diseño de la GUI, deshabilitando esta variable.

Las **funciones** de la clase son:

Invoke	Invoca el método pasado por parámetro en x segundos
InvokeRepeating	Invoca el método pasado por parámetro en x segundos
CancelInvoke	Cancela todas las llamadas a métodos en este **MonoBehaviour**
IsInvoking	Hay alguna invocación de **methodName** pendiente?
StartCoroutine	Empieza un **coroutine**
StopCoroutine	Para todos los **coroutine** llamados **methodName** que están corriendo en este **behaviour**
StopAllCoroutines	Para todos los **coroutine** que están corriendo en este **behaviour**

De las **funciones Overridable** vamos a explicar las funciones más importantes y más utilizadas en este proyecto.

- Update:

  Es la función llamada cada frame, siempre y cuando MonoBehaviour esté habilitado, y su uso más común es para implementar el modo de comportamiento del juego.

- FixedUpdate:

  Esta función es llamada cada fotograma fijo de imágenes por segundo, si el MonoBehaviour está habilitado. Puede resultar interesante su uso, en vez de Update, cuando se trabaja con un rigidbody, si quiere aplicar una fuerza cada fotograma fijo, por ejemplo, hágalo dentro de FixeUpdate, en vez de cada frame dentro de Update.

- Awake:

  Esta función es llamada cuando la instancia del script se está cargando. Suele utilizarse para inicializar variables o el estado del juego antes de que el juego comience.

- Start:

  Es llamada justo antes de que cualquier método Update sea llamado por primera vez. Es llamado solo una vez en la vida del Behaviour.

  La mayoría del resto de funciones como OnMouseEnter, OnMouseOver, OnCollisionEnter, etc, ya han sido explicadas anteriormente por lo que no serán tratadas de nuevo. La diferencia entre Start y Awake es que Start es llamado solamente si la instancia del script está habilitada.

ii) Collider (hereda de Component):

Es la clase fundamental de todos los colliders. Los colliders son objetos que detectan colisiones de otros objetos. Hay cuatro tipos de colliders:

- BoxCollider: Es un collider de forma cuadrangular que posee dos variables para regular su tamaño y posición que son **center** y **size.**
- SphereCollider: Es un collider con forma esférica que tienes dos variables, **center** y **radius** para configurar su tamaño y posición.
- CapsuleCollider: Es un collider con forma de cápsula que tiene como variables de la clase: **center**, **radius**, **height** y **direction**, para configurar la forma.

- MeshCollider: MeshCollider es un collider que se adapta a la forma del objeto al cual se aplica.

La clase collider contiene las siguientes **variables**:

attachedRigidbody	El rigidbody al cual está unido el collider
isTrigger	Es trigger este collider?
material	El material usado por el collider
sharedMaterial	El material para compartir de el collider
bounds	El volumen que delimita el colisionador (collider)

Las **funciones** de la clase:

ClosestPointOnBounds	El punto más cercano la caja de delimitación del collider adjunto
Raycast	Lanza un ray que pasa por alto todos los collider excepto éste

iii) ParticleAnimator (hereda de component):

Este componente es el encargado de mover las partículas en el tiempo, se usa para aplicar viento, fuerzas de arrastre y color a los sistemas de partículas. Esta clase es un script para un componente de animación de partículas.

Las **variables** de la clase son:

doesAnimateColor	Hace ciclar su color durante toda la vida de la particula?
worldRotationAxis	Alrededor de que eje del mundo de las partículas rota
localRotationAxis	Alrededor de que eje local de las partículas rota.
sizeGrow	Como las partículas aumentan el tamaño durante su vida
rndForce	Una fuerza aleatoria es aplicada a las partículas en todo frame
force	La fuerza aplicada a las partículas en todo frame
damping	Cómo de lentas son cada vez más las partículas
autodestruct	Destruye el gameobject de este animador de partículas
colorAnimation	El color de las partículas que utilizarán durante su vida

iv) ParticleEmitter (hereda de component):

Es un script de interfaz para emitir partículas. Tanto ParticleEmitter como ParticleAnimator son scripts que se pueden modificar en el interfaz de Unity en la ventana inspector o en el script.

Las **variables** de la clase son:

emit	Permite a las partículas ser emitidas automáticamente cada frame
minSize	El mínimo tamaño que puede tener cada particula
maxSize	El máximo tamaño que puede tener cada particula
minEnergy	El mínimo tiempo de vida de cada particula
maxEnergy	El máximo tiempo de vida de cada particula
minEmission	El mínimo número de partículas emitidas por segundo
maxEmission	El máximo número de partículas emitidas por segundo
emitterVelocityScale	Aumenta la velocidad de las partículas emitidas
worldVelocity	La velocidad inicial de las partículas en los ejes X, Y, Z globales
localVelocity	La velocidad inicial de las partículas en los ejes X, Y, Z locales
rndVelocity	Una velocidad aleatoria que es añadida sobre los ejes X, Y, Z
useWorldSpace	Si está habilitada, cuando mueves el editor las partículas no se mueven, si está deshabilitada, si que se mueven.
rndRotation	Si está habilitada, las partículas se mueven con rotación aleatoria
angularVelocity	La velocidad angular de las nuevas partículas en grados
rndAngularVelocity	Una nueva velocidad angular que modifica las nuevas partículas
particles	Devuelve una copia de todas las partículas y asigna un array de todas las partículas que están actualmente corriendo.
particleCount	El número de partículas que corren.
enabled	Habilita o deshabilita la emisión de partículas

Las **funciones** son:

ClearParticles	Elimina todas las partículas del sistema
Emit	Emite un numero de partículas
Simulate	Avance del sistema de partículas simulado con tiempo determinado

v) Rigidbody (herada de component):

La clase **Rigidbody** controla la posición de los objetos en una simulación física. Los componentes **Rigidbody** tienen la función de controlar la posición de los objetos, de controlar si al objeto le afectará o no la fuerza gravitatoria, y calcular cómo responden los objetos ante colisiones.

Cuando manipulas los parámetros de un **Rigidbody**, es recomendable trabajar sobre la función **FixedUpdate** como ya comentamos anteriormente.

Un punto a tener en cuenta cuando utilizamos los rigidbodys es:

Si la simulación se observa en cámara lenta y no sólida:

Es un problema de escala. Cuando el mundo del juego es tan grande como parece se mueve muy despacio. Asegúrese de que todos tus modelos se encuentran el tamaño del mundo real. Por ejemplo, si un coche mide aproximadamente 4 metros de largo y en el juego lo ponemos a 2 metros, el objeto mantendrá el peso del objeto de dos metros y la simulación de la caída se corresponderá con la de un coche de dos metros y parecerá que cae a cámara lenta. Por eso es mejor mantener el tamaño de los objetos del mundo real.

Las **variables** de la clase:

velocity	El vector de velocidad del rigidbody
angularVelocity	La velocidad angular del rigidbody
drag	El arrastre
angularDrag	El arrastra angular de el objeto
mass	El peso o masa del objeto
useGravity	Controlas si le afecta o no la gravedad
isKinematic	Controlas si la física afecta al rigidbody
freezeRotation	Controlas si la física afectará al cambio de rotación del objeto
constraints	Controla qué grados de libertad son permitidos en la simulación del ese Rigidbody
collisionDetectionMode	El modo de detección de las colisiones
centerOfMass	El centro de masa relativo al transfrom original
worldCenterOfMass	El centro de masa del rigidbody en el world space
inertiaTensorRotation	La rotación del tensor de inercia
inertiaTensor	El tensor diagonal de inercia de masa relativa para el centro de masa
detectCollisions	Está habilitada la detección de colisiones?
useConeFriction	La fricción de la Fuerza de cono que se utilizará para este rigidbody
position	La posición del rigidbody

rotation	La rotación del rigidbody
interpolation	Te permite suavizar el efecto de la física corriendo a una velocidad fija
solverIterationCount	Le permite anular el número de iteraciones solucionadas por rigidbody
sleepVelocity	La velocidad linear, por debajo del cual los objetos comienzan a dormir
sleepAngularVelocity	La velocidad angular, por debajo la cual los objetos comienzan a dormir
maxAngularVelocity	La máxima velocidad angular de un rigidbody

Las **funciones** son:

SetDensity	Establece la masa sobre la base de los colliders adjunta asumiendo una densidad constante.
AddForce	Añade fuerza a el rigidbody
AddRelativeForce	Añade una fuerza relativa a las coordenadas del sistema del rigidbody
AddTorque	Añade un torque al rigidbody
AddRelativeTorque	añade un torque al rigidbofy relativo a las coordenadas del sistema
AddForceAtPosition	Añade una fuerza en una posición
AddExplosionForce	Simula una explosión y aplica la fuerza correspondiente al rigidbody
ClosestPointOnBounds	El punto más cercano a la caja de detección de colisiones
GetRelativePointVelocity	La velocidad del rigidbody relativa al punto relativePoint
GetPointVelocity	La velocidad de un rigidbody en un punto del world space
MovePosition	Mueve el rigidbody a position
MoveRotation	Rota el rigidbody a rotation
Sleep	Fuerza al rigidbody a dormir por lo menos un frame
IsSleeping	Está durmiendo?
WakeUp	Fuerza al rigidbody a despertar
SweepTest	Comprueba si rigidbody ha colisionado con alguien
SweepTestAll	Como el anterior, pero devuelve todos los golpes.

Esta clase tiene las funciones OnCollisionEnter, OnCollisionExit y OnCollisionStay, para enviar mensajes a otros objetos sobre las colisiones que hayan tenido.

vi) Transform (hereda de Component y IEnumerable):

Todo objeto en una escena tiene un componente Transform. Es usado para almacenar y manipular la posición, rotación y escala de los objetos. Todo Transform puede tener

un parent (padre), que te permite cambiar y aplicar la posición, rotación y la escala jerárquicamente, en el panel de Jerarquía (Hierarchy).

Las **variables** son:

position	La posición del transform en el world space
localPosition	La posición del transform en el local space
eulerAngles	La rotación en grados Euler
localEulerAngles	La rotación en grados Euler relativos a la rotación del padre
right	El eje rojo en el transform world space
up	El eje verde en el transform world space
forward	El eje azul en el transform world space
rotation	La rotación del el transform en el world space almacenada como un Quaternion
localRotation	La rotación del transform relativa a la rotación del padre
localScale	La escala del transform relativa al padre
parent	El padre del transform
worldToLocalMatrix	Matriz que transforma un punto del world space dentro de local space
localToWorldMatrix	Inverso que la anterior
root	Devuelve transform raíz de los transform que hay en Hierarchy
childCount	El número de hijos de transform que hay.
lossyScale	La escala global del objeto

Las **funciones** son:

Translate	Mueve el transform en dirección y distancia de traslación
Rotate	Aplica la rotación
RotateAround	Rota alrededor de un punto de coordenadas
LookAt	Rota el transform de modo que el vector apunte hacia adelante a un objeto
TransformDirection	Transforma la dirección del local space a world space
InverseTransformDirection	Inversa que la función anterior
TransformPoint	Transforma la posición del local space a world space
InverseTransformPoint	La inversa que la función anterior
DetachChildren	Quitar el padre a todos los hijos
Find	Encuentra un hijo por nombre y lo devuelve
IsChildOf	Es este transform hijo del padre?

b) GameObject (hereda de Object):

Esta clase es la clase plantilla para todas las entidades de las escenas en Unity.

Las **variables** son casi las mismas que la clase **component**:

isStatic	Especifica si el game Object es estatico o no
transform	El Transform que está unido al GameObject (null si no hay unido)
rigidbody	El Rigidbody que está unido al GameObject (null si no hay unido)
camera	El Camera que está unido al GameObject (null si no hay unido)
light	El Light que está unido al GameObject (null si no hay unido)
animation	El Animation que está unido al GameObject (null si no hay unido)
constantForce	El ConstantForce que está unido al GameObject (null si no hay unido)
renderer	El Renderer que está unido al GameObject (null si no hay unido)
audio	El AudioSource que está unido al GameObject (null si no hay unido)
guiText	El GUIText que está unido al GameObject (null si no hay unido)
networkView	El NetworkView que está unido al GameObject (null si no hay unido)
guiTexture	El GUITexture que está unido al GameObject (null si no hay unido)
collider	El Collider que está unido al GameObject (null si no hay unido)
hingeJoint	El HingeJoint que está unido al GameObject (null si no hay unido)
particleEmitter	El ParticleEmitter que está unido al GameObject (null si no hay unido)
gameObject	El game object al que están unidos los componentes
tag	El tag del game object.

El constructor de la clase es **GameObject,** que crea un nuevo objeto llamado name (nombre pasado por parámetro).

Las **funciones** de la clase:

GetComponent	Devuelve el componente del tipo o del nombre pasado por parámetro si el objeto tiene unido alguno, si no, devuelve null
GetComponentInChildren	Devuelve el componente del tipo seleccionado del gameObject o de alguno de sus hijos usando la búsqueda en profundidad
GetComponentsInChildren	Devuelve los componentes del tipo seleccionado del GameObject o alguno de sus hijos
GetComponents	Devuelve todos los componentes del tipo seleccionado que tiene el GameObject
SetActiveRecursively	Cambia el estado a activo de todos los hijos del gameObject

CompareTag	Este GameObject está etiquetado como tag?
SendMessageUpwards	Invoca al método llamado methodName en cualquier script de ese gameObject y en todos los ancestros de el gameObject
SendMessage	Invoca al método llamado methodName en cualquier script de ese gameObject
BroadcastMessage	Invoca al método llamado methodName en cualquier script de ese gameObject y todos sus hijos.
AddComponent	Añade un componente de la clase className al game object
SampleAnimation	Muestra una animación en un momento dado de las propiedades de animación.

Funciones de la clase:

CreatePrimitive	Crea un game object con una textura de renderizado y su collider asociado
FindWithTag	Devuelve el GameObject activo etiquetados con tag.
FindGameObjectsWithTag	Devuelve una lista de GameObjects etiquetados con tag
Find	Busca un game object por nombre (name)

c) Texture (hereda de Object):

Es la clase base para el manejo de las texturas. Contiene funcionalidades que son comunes en las clases Texture2D y RenderTexture.

Las **variables** de la clase son:

width	La anchura de la textura en pixeles
height	La altura de la textura en pixeles
filterMode	El modo de filtro aplicado a la textura
anisoLevel	Nivel de Anisotropic filtro para la textura
wrapMode	Envuelve el tipo de la textura (Repeat o Clamp)
mipMapBias	Mip map bias de la textura

La clase contiene la función **GetNativeTextureID** que recupera el “hardware” nativo para manejar una textura.

7) Ray (estructura):

Es la representación de los rays. Un ray es una línea infinita que comienza en **origin** (origen) y va en cualquier dirección.

Las **variables** son **origin** (el punto de origen del **ray**) y **direction** (la dirección del ray).

El constructor es **Ray**, crea un ray que comienza en origin hacia direction.

Las funciones son GetPoint, que devuelve el punto en las unidades de distancia a lo largo del rayo, y ToString que devuelve una cadena (string) sobre el rayo con un formato entendible.

8) RaycastHit (estructura):

Es una estructura usada para obtener información sobre un raycast.

Las **variables** son las siguientes:

point	El punto de impacto sobre el world space donde el ray golpea al collider
normal	La normal que sobre sale del golpe del ray
barycentricCoordinate	La coordenada barycentric del triángulo que golpea
distance	La distancia del origen al punto de impacto
triangleIndex	El índice del triángulo que ha sido golpeado
textureCoord	La textura uv que coordina el punto de impacto
textureCoord2	La textura uv secundaria que coordina el punto de impacto
lightmapCoord	El uv lightmap que coordina el punto de impacto
collider	El collider que ha sido golpeado
rigidbody	El Rigidbody del collider que ha sido golpeado
transform	El transform del rigidbody o el collider que ha sido golpeado

9) Screen:

Es usada para obtener información sobre la pantalla, una lista de resoluciones soportadas, la resolución actual, etc.

Las **variables** de la clase son:

resolutions	Todas las resoluciones soportadas por el monitor
currentResolution	La actual resolución
showCursor	Mostramos el cursor?
lockCursor	Bloqueamos el cursor?
width	La anchura actual de la pantalla en píxeles
height	La altura actual de la pantalla en píxeles
fullScreen	Está el juego utilizando toda la pantalla?
orientation	Especifica la orientación lógica de la pantalla

La función de la clase es **SetResolution** con la cual puedes cambiar de resolución a tu juego.

10) Time:

El interfaz para obtener información sobre el tiempo en Unity.

Las **variables** de la clase:

time	El tiempo en el cual el frame ha comenzado. Se cuenta en segundos desde que el juego ha comenzado
timeSinceLevelLoad	El tiempo en el cual el frame ha comenzado. Se cuenta desde que el último nivel ha sido cargado
deltaTime	El tiempo en segundos que tardó en completar el último frame
fixedTime	El tiempo en el cual el último FixedUpdate ha empezado
fixedDeltaTime	El intervalo de tiempo en el cual physics y otros fixed frame rate updates se llevan a cabo
maximunDeltaTime	El tiempo máximo que un frame puede tomar
smoothDeltaTime	A smoothed out Time.deltaTime
timeScale	La escala en la que se pasa el tiempo
frameCount	EL número total de frames que han sido pasados
realtimeSinceStartup	El tiempo real en segundos desde que el juego ha comenzado
captureFramerate	Si captureFramerate es inicializado con una valor mayor que 0 el tiempo avanza en x.

11) Touch (estructura):

Es una estructura en la que se describe el estado del toque de un dedo en la pantalla.

Las **variables** son:

fingerId	El identificador único del touch
position	La posición del touch
deltaPosition	La posición delta desde el último cambio
deltaTime	Acumulación de tiempo pasado desde el último cambio
tapCount	Numero de taps
phase	Describe la fase del touch

12) Vector3 (estructura):

Es la representación de vectores 3D y puntos.

Esta estructura es usada para pasar la posición 3D y la dirección.

Las variables:

x	El componente x del vector
y	El componente y del vector

z	El componente z del vector
this[int index]	Accede a los componentes x, y, z utilizando [0], [1], [2] respectivamente
normalized	Devuelve el vector con magnitud de 1
magnitude	Devuelve la extensión del vector
sqrMagnitud	Devuelve la longitud al cuadrado de este vector

El **constructor** de la clase es Vector3, crea un nuevo vector con las componentes x, y, z.

Las **funciones** son:

Scale	Multiplica todos los componentes por el mismo componentes de escala
Normalize	Crea un vector con magnitud 1
ToString	Devuelve una cadena sobre el vector

Las **variables** de la clase:

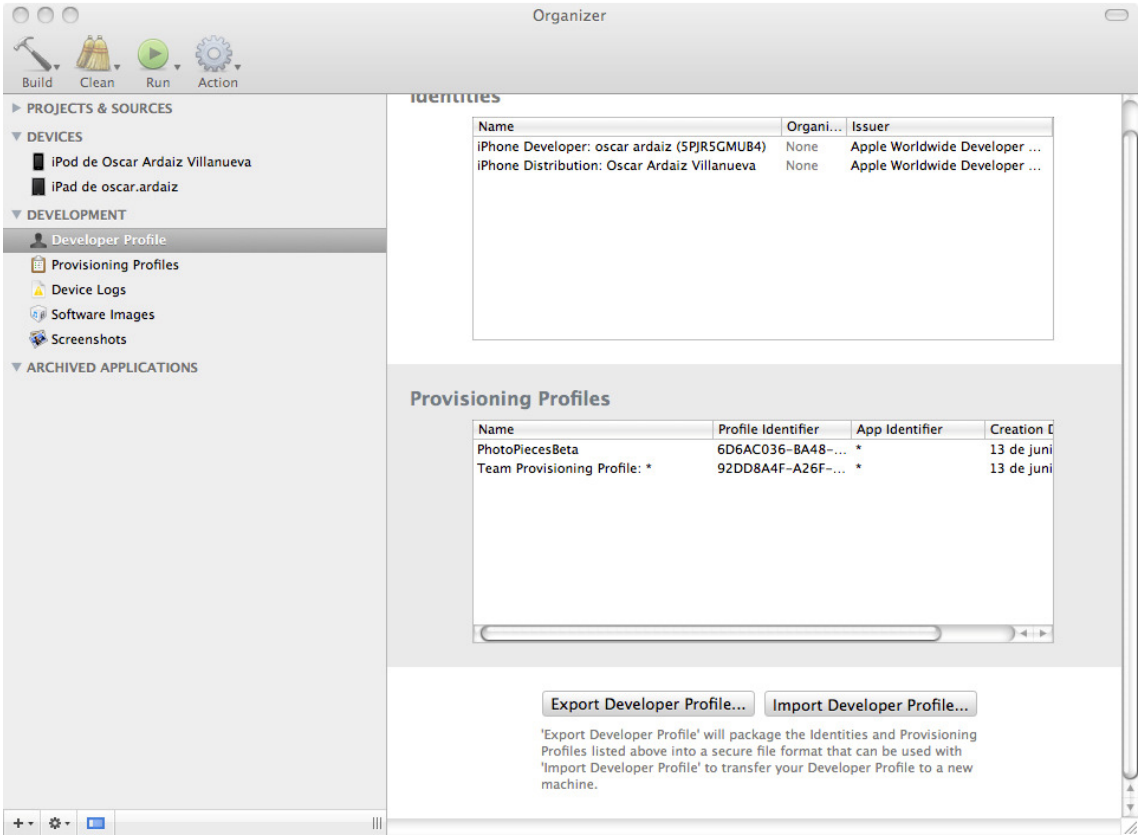
zero	Crea el vector Vector3(0, 0, 0)
one	Crea el vector Vector3(1, 1, 1)
forward	Crea el vector Vector3(0, 0, 1)
up	Crea el vector Vector3(0, 1, 0)
right	Crea el vector Vector3(1, 0, 0)

La clase contiene otras funciones que se utilizan para operar entre vectores, como en nuestro proyecto no hemos utilizado estas funciones no las vamos a explicar.

- **Método de compilación y ejecución de la aplicación en el iPad:**

Una vez vistas y explicadas cada una de las clases utilizadas en la aplicación, vamos a explicar el proceso a seguir para que nuestra aplicación corra en el dispositivo iPad.

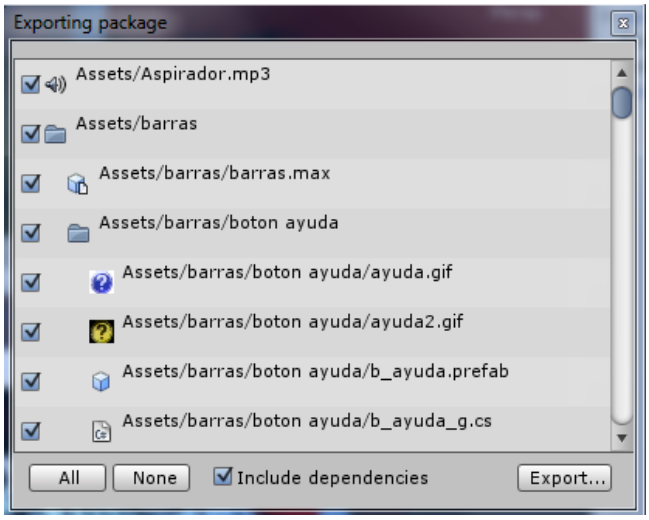
Lo primero de todo es comunicar a la empresa Apple que vas a realizar aplicaciones para el dispositivo iPad para conseguir la licencia. Esta licencia se llama "Apple Developer profile". A parte de esto, debes registrar los dispositivos para los cuales se va a desarrollar aplicaciones, en nuestro caso, deberíamos registrar el dispositivo iPad.



El siguiente paso, si usted no posee un Mac y ha construido la aplicación sobre Windows, debe exportar todos los paquetes creados para después importarlos sobre el equipo Mac, para en adelante poder pasar la aplicación al iPad.

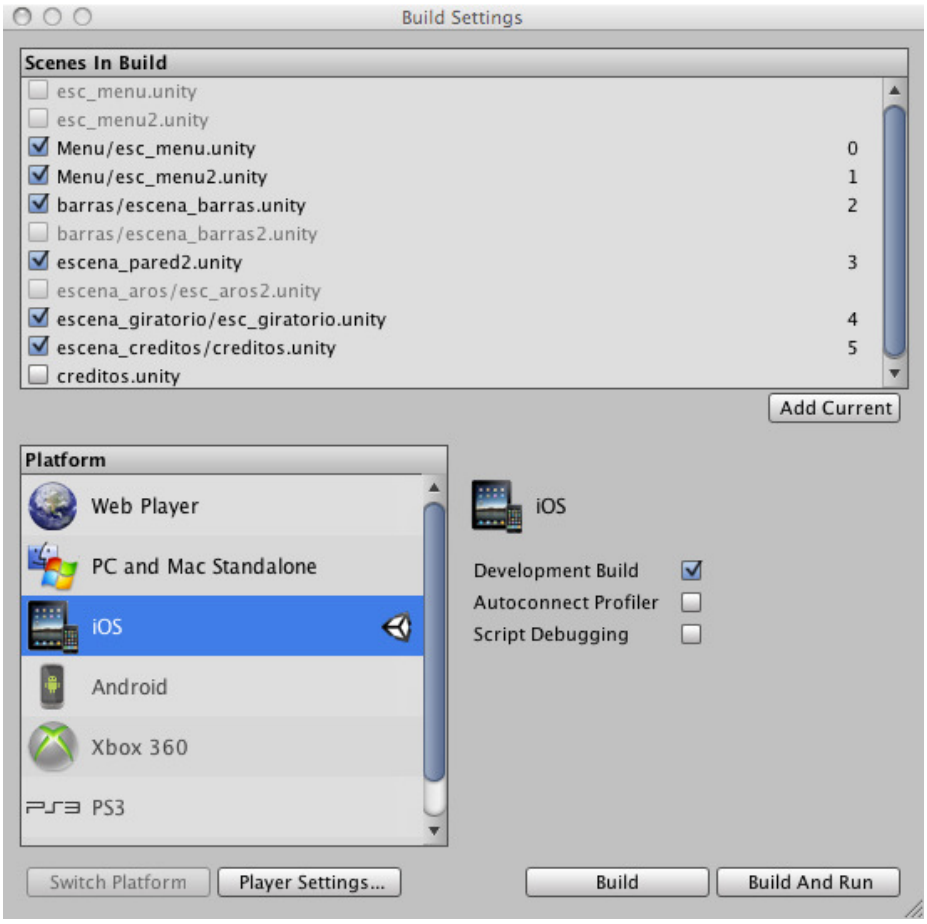
Para exportar los paquetes debes seleccionar todos los elementos de la vista Project. Esto se hace seleccionando en el menú **Edit->Select All**, o bien cuando tienes seleccionado un objeto de la vista Project presionar las teclas **Ctrl+A.**

Una vez seleccionados todos los elementos, seleccionar **Assets->Export Package**…, a continuación aparecerá una lista con todos los objetos seleccionado para exportar en la cual si seleccionas la opción Export…, deberás elegir el directorio para guardar el package.



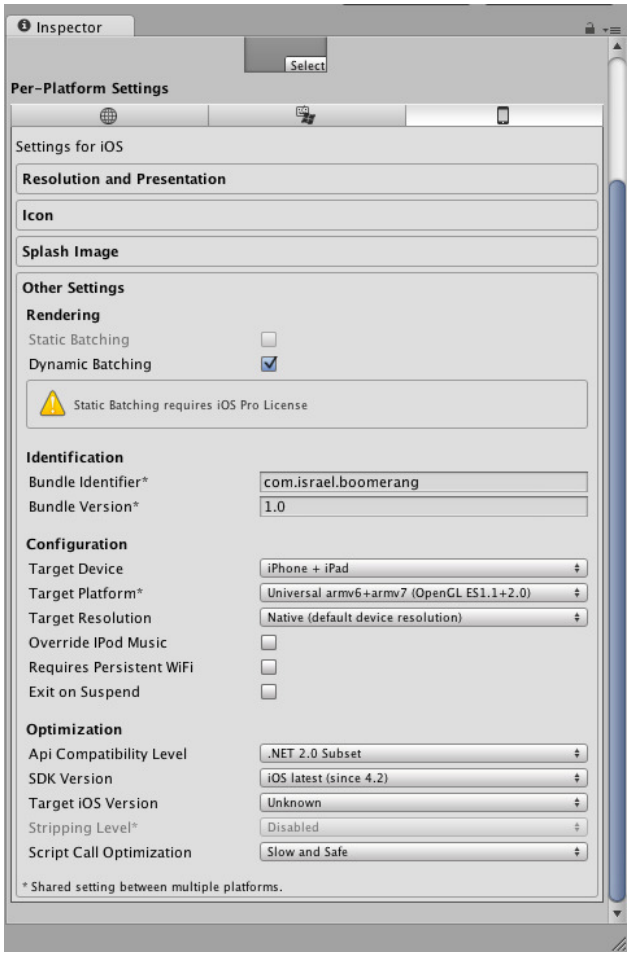
El proceso de importar el paquete, es precisamente el contrario, consiste en abrir Unity en el Mac, seleccionar "**Assets->Import Package…"** y seleccionas el archivo resultante de la operación anterior.

Cuando ya tienes todo el archivo importado, suele suceder que las escenas de la aplicación no están montadas. Por lo que debemos ir a la opción **File->BuildSettings.** Al pulsar esta opción aparece la siguiente ventana:



Para cargar las escenas, solo hay que abrir la escena en Unity y dar a la opción "**Add Current**"

En esta ventana (BuildSettings), se configura también la plataforma sobre la cual va a correr nuestra aplicación. Seleccionamos iOS ya que nuestra aplicación queremos que funciones sobre un iPad. Y para configurar los parámetros de construcción seleccionamos la opción **"Player Settings…".**



Al seleccionar esta opción, aparece en la vista inspector, varios atributos para cambiar sobre como renderizar y construir nuestro proyecto.

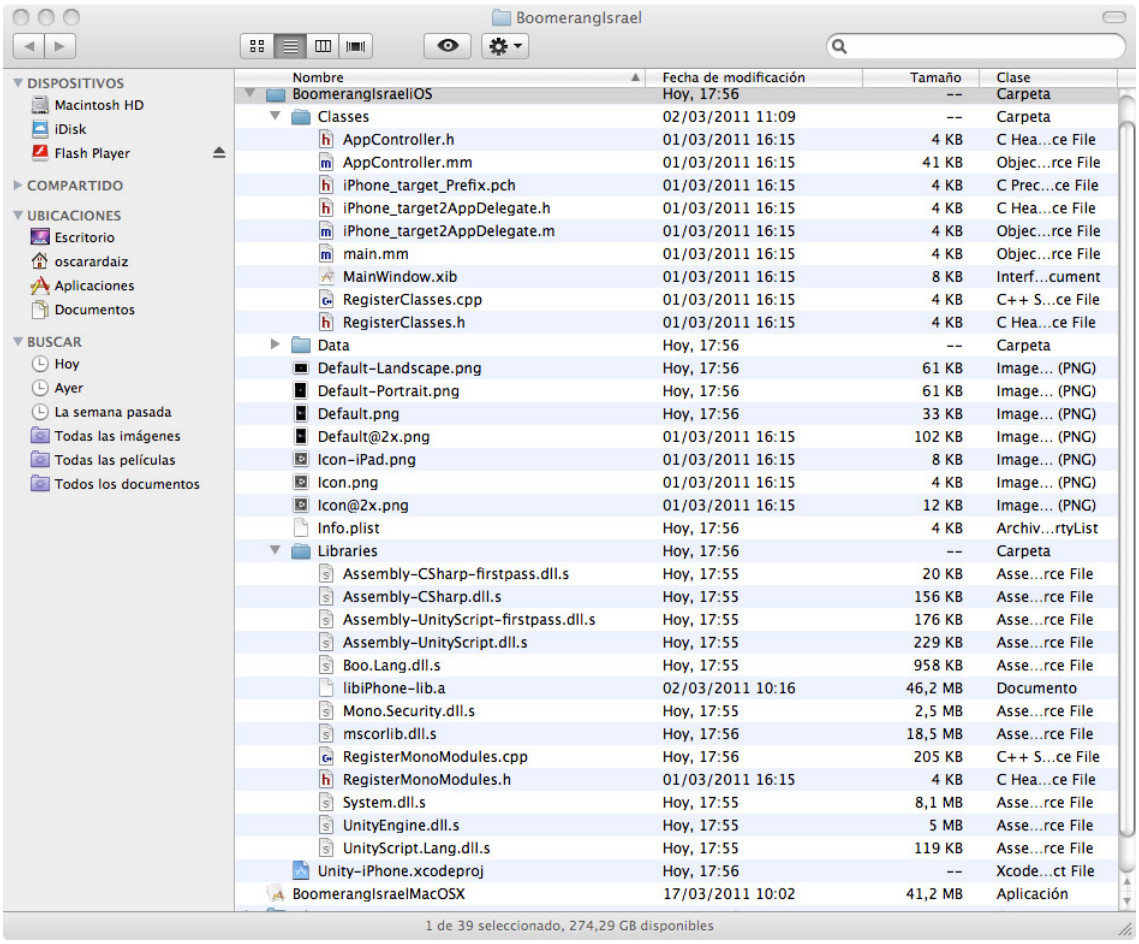
También se configura el dispositivo destino, la plataforma, la versión SDK, y muchas opciones que nosotros dejamos por defecto.

Una vez configurado todo esto, seleccionamos en la ventana de BuildSettings, la opción Build y seleccionamos un directorio destino.

El propio Unity se encargar de transformar el lenguaje C#, propiedad de Windows, a un lenguaje que pueda ser entendido por el dispositivo iPad.

Mono, es un proyecto libre y de código abierto liderado por Xamarin compatible con el estándar .NET que proporciona un conjunto de herramientas, que incluyen, entre otros, un compilador de C#. El propósito de "mono" no es solo poder ejecutar .NET en muchas plataformas, sino también gestionar mejor las herramientas de desarrollo para los desarrolladores de Linux. Mono se puede ejecutar sobre Android, BSD, IOS, Linux, Mac OS X, Windows, Solaris y Unix, y también sobre las videoconsolas PS3, Wii y Xbox 360.

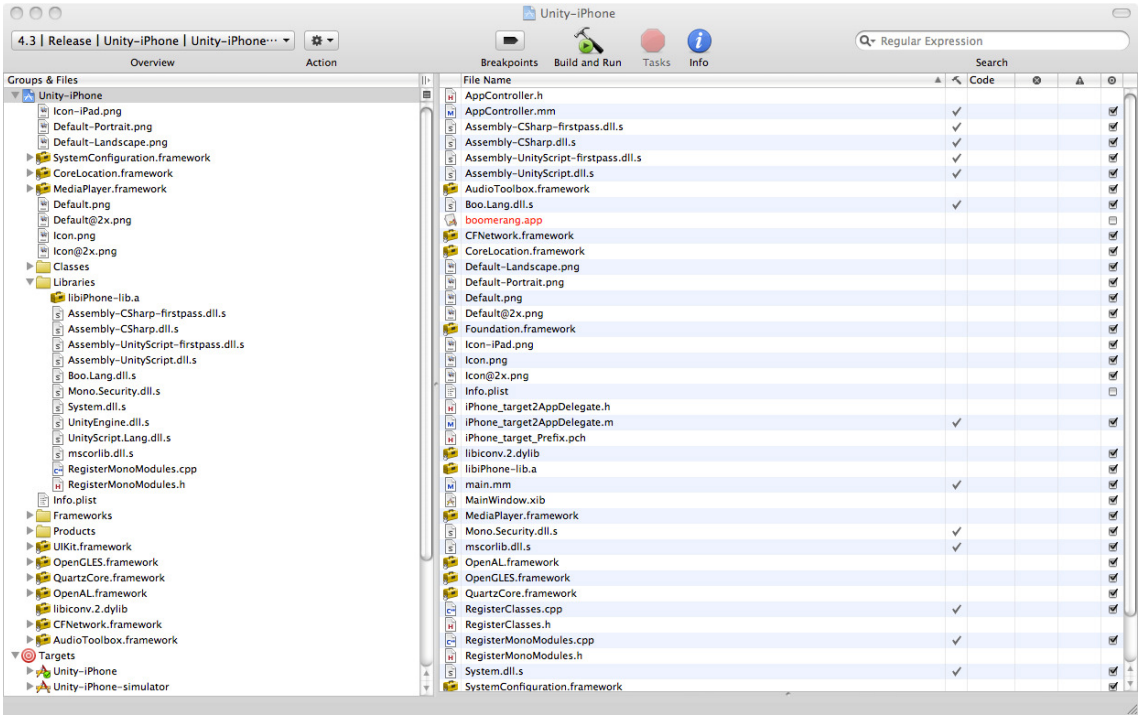
Cuando termina de contruir Unity el proyecto, se crean una gran cantidad de archivos. Veamos la siguiente imagen:



Después de esto, se deben de compilar los archivos con el compilador Xcode que se suministra gratuitamente junto con Mac OS X.

Como se observa en la siguiente imagen, aparecen todos los archivos del proyecto que se deben compilar, si no aparecieran deberíamos seleccionar nuestro proyecto para después compilarlo.

Para realizar el último paso, debemos conectar el iPad al portátil, y sobre el compilador Xcode seleccionar la opción de Build and Run. El compilador, realizará las operaciones necesarias y cuando termine el juego se ejecutará en el iPad, siempre y cuando no dé ningún tipo de error en la compilación.



3.3.Análisis:

• Análisis de requisitos:

En la fase de estudio del software Social, observamos ciertas características que marcarían la forma de construir nuestra aplicación. Como nuestra aplicación iba a constar de varios niveles en cuanto a las pruebas de lanzamiento de un boomerang, se decidió realicar un menú con el cual, podrías jugar desde la primera prueba o bien seleccionar la pantalla deseada.

Vamos a ver los requisitos propios de cada escena:

Escenas del **menú**:

- El menú constará de dos escenas, con opciones diferentes.
- Las opciones de la primera escena serán las de jugar, ver los créditos y salir del juego.
- Las opciones de la segunda escena serán las de acceder a las escena del boomerang, a la del muro, la del giratorio, y la opción de poder regresar al menú anterior.
- La forma de presionar los botones será táctil, con un toque será suficiente para acceder a las opciones
- Los menús serán en tres dimensiones para aprovechar las facilidades en ese sentido de la aplicación Unity.

Escena de **barras** "**boomerang**":

- La prueba consistirá en pasar el boomerang primero por las barras de la derecha y luego por las de la izquierda, en caso de hacerlo en sentido contrario no contará como válido el lanzamiento.
- Para lanzar el boomerang, habrá que pulsar un botón con forma de boomerang de forma continuada, de tal forma que cuanto más tiempo se encuentre pulsado más velocidad obtendrá el boomerang.
- La escena debe de poseer dos botones, uno para recibir las instrucciones de la pantalla y otro, para parar el juego y poder salir.
- El tiempo para conseguir tres lanzamientos correctos o válidos es de 60 segundos.
- No se podrá lanzar un boomerang mientras otro se encuentra en movimiento.
- Cuando aparezca un menú, de salir, de ayuda, de tiempo agotado o de prueba conseguida, no se podrá tener acceso a los botones de salir, ayuda y disparar.
- Para girar la cámara o vista, se deberá arrastrar el dedo sobre la pantalla hacia la dirección deseada.

- Para cambiar de posición y moverse por la pantalla, se utilizará la característica del acelerómetro, de tal modo que inclinando para un lado el dispositivo el personaje se mueve para ese lado, y al revés.

Escena del **Muro:**

- La prueba consistirá en pasar el boomerang primero por la ventana de la derecha y luego por la de la izquierda, en caso de hacerlo en sentido contrario no contará como válido el lanzamiento.
- Para lanzar el boomerang, habrá que pulsar un botón con forma de boomerang de forma continuada, de tal forma que cuanto más tiempo se encuentre pulsado más velocidad obtendrá el boomerang.
- La escena debe de poseer dos botones, uno para recibir las instrucciones de la pantalla y otro, para parar el juego y poder salir.
- El tiempo para conseguir tres lanzamientos correctos o válidos es de 60 segundos.
- Por cada lanzamiento, la dificultad se incrementará, de modo que cuando pase el primer lanzamiento el muro comenzará a moverse, y cuando pase el segundo el muro, duplicará su velocidad.
- No se podrá lanzar un boomerang mientras otro se encuentra en movimiento.
- Cuando aparezca un menú, de salir, de ayuda, de tiempo agotado o de prueba conseguida, no se podrá tener acceso a los botones de salir, ayuda y disparar.
- Para girar la cámara o vista, se deberá arrastrar el dedo sobre la pantalla hacia la dirección deseada.
- Para cambiar de posición y moverse por la pantalla, se utilizará la característica del acelerómetro, de tal modo que inclinando para un lado el dispositivo el personaje se mueve para ese lado, y al revés

Escena del **Giratorio:**

- La prueba consistirá en golpear todas las barras del giratorio de manera que para conseguir el objetivo todas las barras deberán estar en dorado.
- Cuando una barra sea golpeada con el boomerang cambiará la textura plateada por la dorada.
- Para lanzar el boomerang, habrá que pulsar un botón con forma de boomerang de forma continuada, de tal forma que cuanto más tiempo se encuentre pulsado más velocidad obtendrá el boomerang.
- La escena debe de poseer dos botones, uno para recibir las instrucciones de la pantalla y otro, para parar el juego y poder salir.

- El tiempo para conseguir el objetivo es de 60 segundos.
- La base del giratorio se encontrará durante toda la prueba en movimiento.
- No se podrá lanzar un boomerang mientras otro se encuentra en movimiento.
- Cuando aparezca un menú, de salir, de ayuda, de tiempo agotado o de prueba conseguida, no se podrá tener acceso a los botones de salir, ayuda y disparar.
- Para girar la cámara o vista, se deberá arrastrar el dedo sobre la pantalla hacia la dirección deseada.
- Para cambiar de posición y moverse por la pantalla, se utilizará la característica del acelerómetro, de tal modo que inclinando para un lado el dispositivo el personaje se mueve para ese lado, y al revés

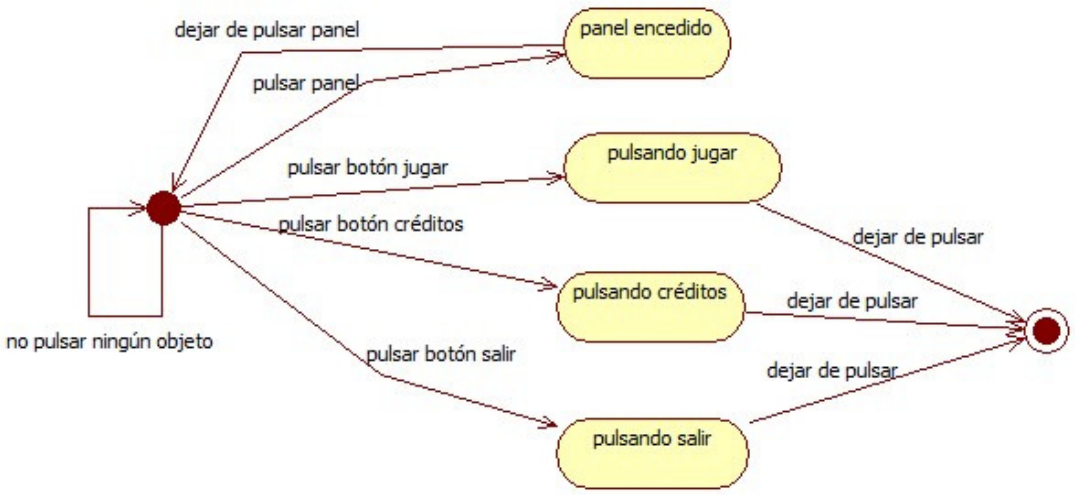
Escena de **Créditos**:

- Es una escena informativa en la que deberá aparecer el eslogan del juego y el creador de la aplicación de la forma más atractiva posible.
- La escena deberá dar opción a regresar al menú.

Una vez analizados los requisitos realizamos los diagramas de estado para representar como sería el funcionamiento de cada escena.

• Diagramas de estado de la aplicación:

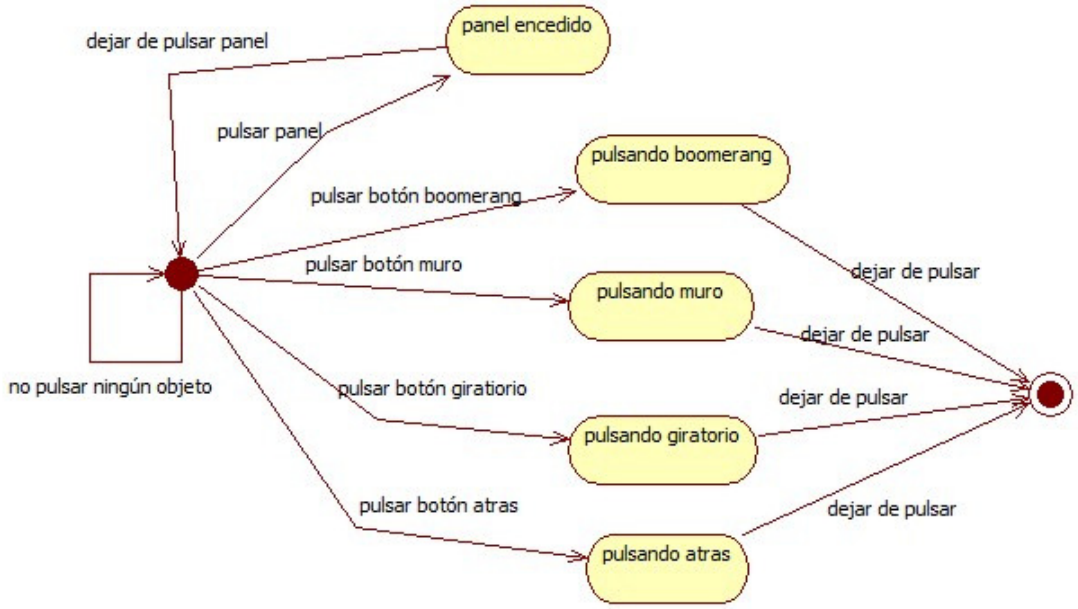
Diagrama de estado de la escena del menú "esc_menu":



La escena correspondiente a este diagrama, es un menú con las opciones de Jugar, acceder a los créditos de la aplicación y la opción de salir de la aplicación. E de recordar, que es una aplicación destinada para el uso con un iPad, por lo que el interfaz y el modo de uso están enfocados para la facilidad propia de una aplicación

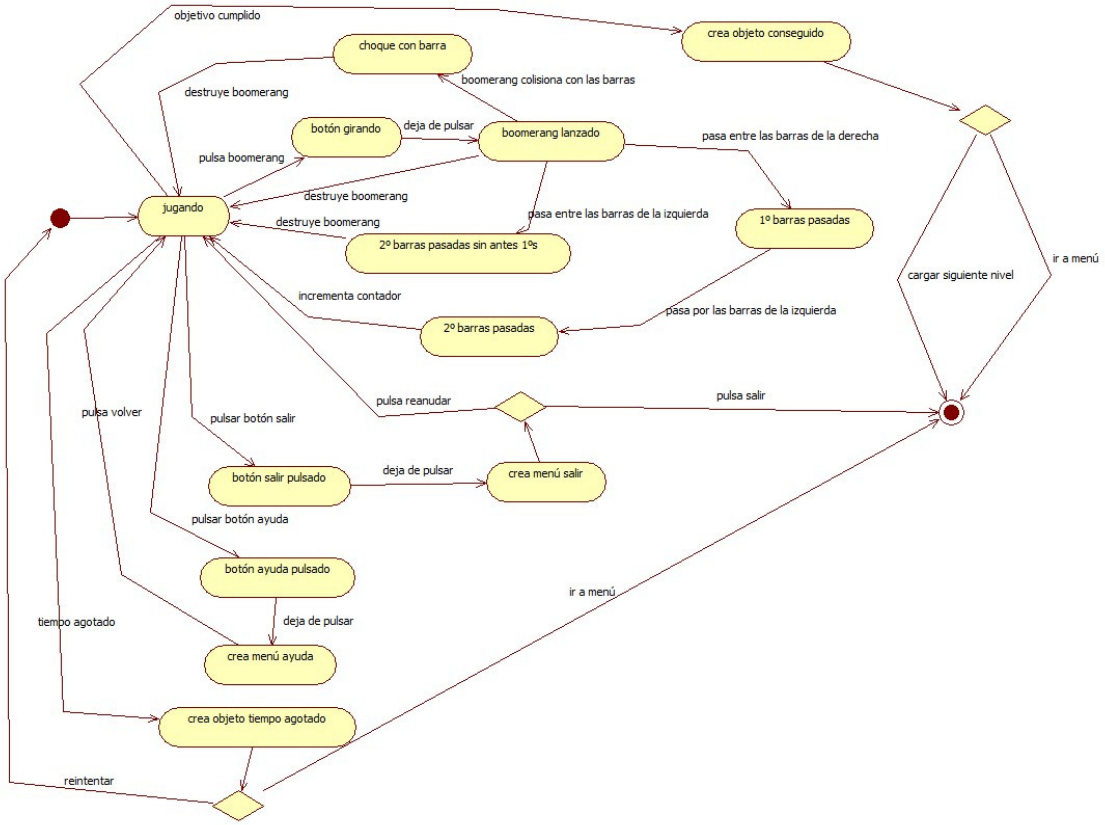
iPad. El modo de interactuar con la aplicación es mediante una pantalla táctil. Si el usuario pulsa el objeto botón Jugar, accede a otra escena de menú para seleccionar que juego desea usar. En el caso del botón de Créditos accede a la escena de créditos, y en el caso de pulsar la opción de Salir, la aplicación es cerrada. La escena contiene un panel que se ilumina cada vez que es pulsado sobre la pantalla.

Diagrama de estado de la escena del segundo menú "esc_menu2":



En cuanto a la "esc_menu2", el funcionamiento y la usabilidad son prácticamente igual a la escena anterior, es la segunda pantalla de menú de nuestra aplicación y para mantener la línea del menú, esta escena comparte varios componentes de la anterior escena. Las opciones de menú de esta escena son acceder a la pantalla Boomerang, acceder a la escena Muro, acceder a la pantalla Giratorio o retroceder en el menú pulsando el botón Atrás. Esta escena también contiene el panel.

Diagrama de estado de la pantalla Boomerang "esc_barras":



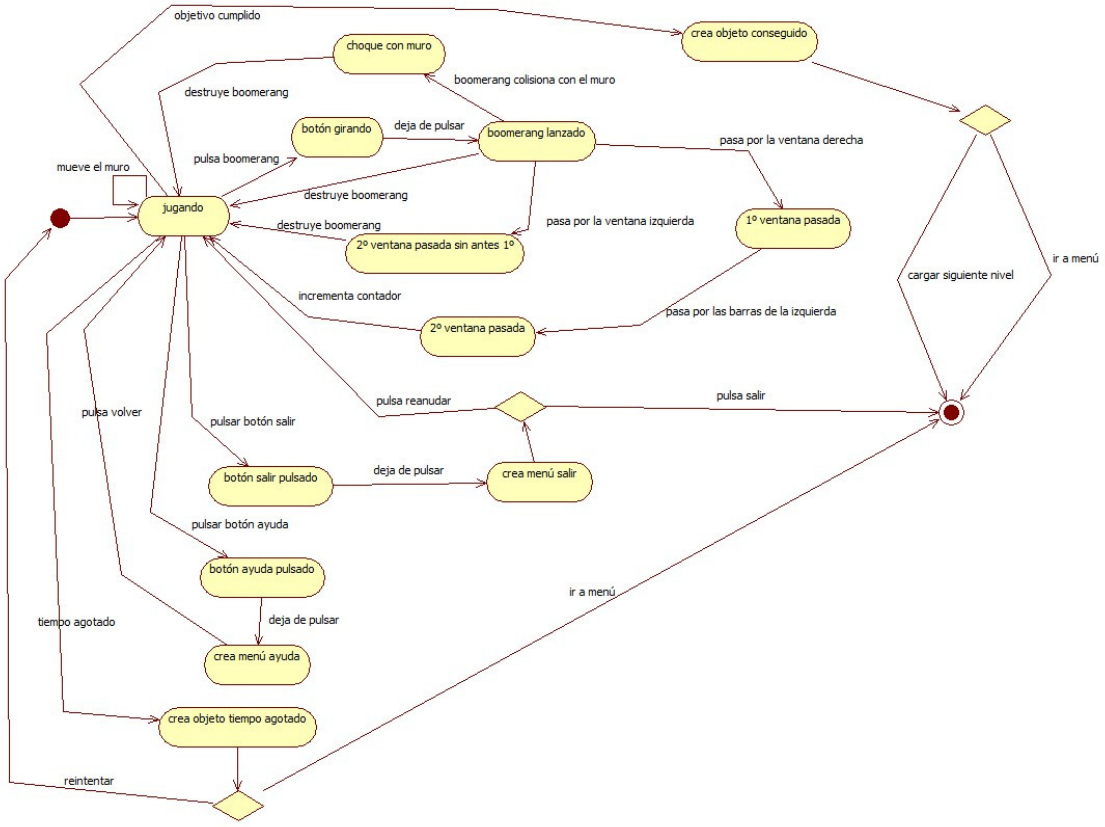
Esta escena ya corresponde más al juego y no tanto al menú de acceso de la aplicación. En la imagen se observa que del estado de inicio directamente pasa al estado "jugando". Esto se debe a que la escena comienza e inicializa numerosos valores como, por ejemplo, el contador, y crea los objetos necesarios para que pueda el usuario jugar, como los botones de ayuda y salir. Una vez inicializadas las variables necesarias, la escena accede al estado jugando que será el encargado de controlar el transcurso del juego, de este modo si el usuario consigue el objetivo del juego, directamente pasa al estado "crea objeto conseguido", que como su nombre indica crea un objeto con una textura en el cual indica que ha conseguido el objetivo de la pantalla, y, además, da las opciones de "cargar el siguiente nivel" e "ir al menú". En el caso de que la opción seleccionada se "cargar el siguiente nivel", cargará la escena del "Muro", mientras que si la opción seleccionada sea "ir a menú", carga la escena de menú principal, "esc_menu". La escena contiene un objeto boomerang centrado abajo en la pantalla, que si es pulsado el objeto comienza a girar, y al ser soltado da como resultado el lanzamiento del boomerang que realiza un movimiento en espiral simulando el movimiento habitual de los boomerangs. En realidad, el objeto boomerang que es pulsado, al dejar de ser pulsado se deshabilita desapareciendo de la pantalla, y otro objeto boomerang (de diferente tipo que el botón) es instanciado e impulsado con unas fuerzas configuradas de tal manera que generen el efecto de elipse propia de los boomerangs. Cuando este objeto es destruido, bien por un choque con las barras o por

que ha completado la elipse, se detecta que ya no existe un objeto en la escena y se habilita el botón boomerang. La manera de controlar la fuerza del boomerang, es presionando más o menos tiempo el botón del boomerang; cuanto más tiempo pulsado más fuerza. Una vez lanzado si el boomerang cruza entre las barras, se instancia un objeto de emisión de partículas para indicar que ha pasado entre las barras el boomerang. Estas partículas se mantendrán en escena mientras el boomerang este realizando el movimiento de elipse. Si el boomerang pasa las barras de la derecha, éstas comunican a las barras de la izquierda que el boomerang ha pasado, para así, si el boomerang consigue pasar también por las barras de la derecha, notificar que el lanzamiento del boomerang ha sido bueno e incrementar el contador en 1, para llegar a conseguir el objetivo de la escena (pasar tres boomerangs).

La escena contiene dos botones para acceder a los menús de ayuda y salir. Cada botón es una textura 2D que detecta si ha sido pulsada, para intercambiar de textura y dar la sensación de botón. Una vez soltado el botón, se crea el menú. En el caso del botón de ayuda, se crea un objeto que contiene una textura 2D informativa sobre cómo jugar y otra textura 2D, para poder volver a jugar y salir del menú de ayuda. Mientras que en el caso del botón de salir, se crea un objeto que contiene una textura 2D con un mensaje de advertencia, y otras dos texturas con las opciones de reanudar la partida y salir.

El estado jugando también controla el tiempo de la partida, y en el caso de que el tiempo se agote pasa automáticamente al estado de “crea objeto tiempo agotado”. En este estado se crea un objeto con una textura informando sobre que el tiempo se ha agotado y dos opciones para poder reintentar el nivel, lo cual reinicia la escena, y para poder acceder al menú presionado “ir a menú”.

Diagrama de estado de la pantalla Muro "escena_pared2":

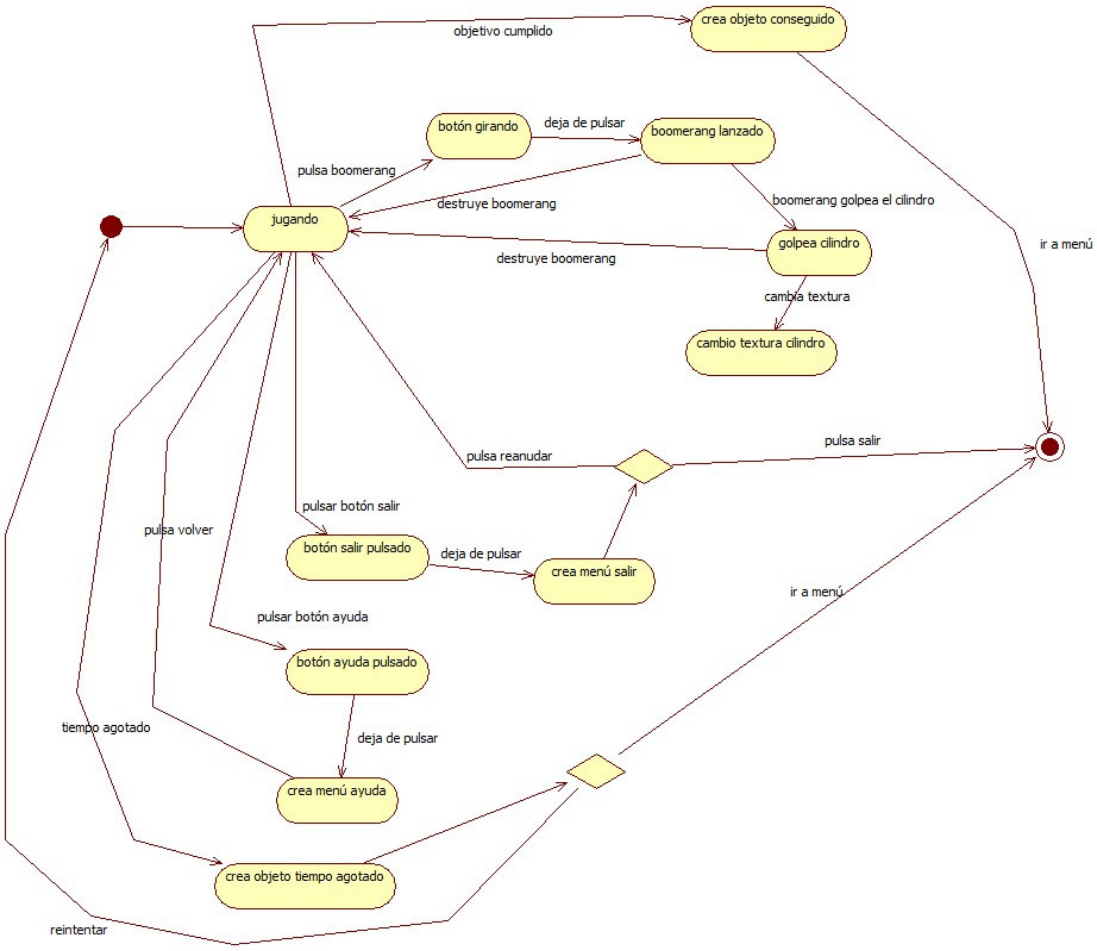


El objetivo principal de esta escena es pasar el boomerang primero por la ventana derecha y luego por la izquierda en un lanzamiento hasta conseguir tres lanzamientos que cumplan ese requisito.

El mecanismo para lanzar el boomerang es el mismo al nombrado en la anterior escena, ya que comparten objetos y código ambas escenas. A diferencia de la escena anterior, en esta aparece un muro con dos ventanas, y una vez lanzado el boomerang si éste choca con el muro produce una pequeña explosión, al igual que ocurre cuando sucedía la colisión en una barra. El funcionamiento de los botones de salir y ayuda es el mismo que en la anterior escena.

En el caso de que hayas pasado un boomerang correctamente entre las ventanas, el muro comenzará a moverse para complicar el nivel. El próximo lanzamiento correcto, duplicará la velocidad del muro.

Diagrama de estado de la pantalla Giratorio "esc_giratorio":



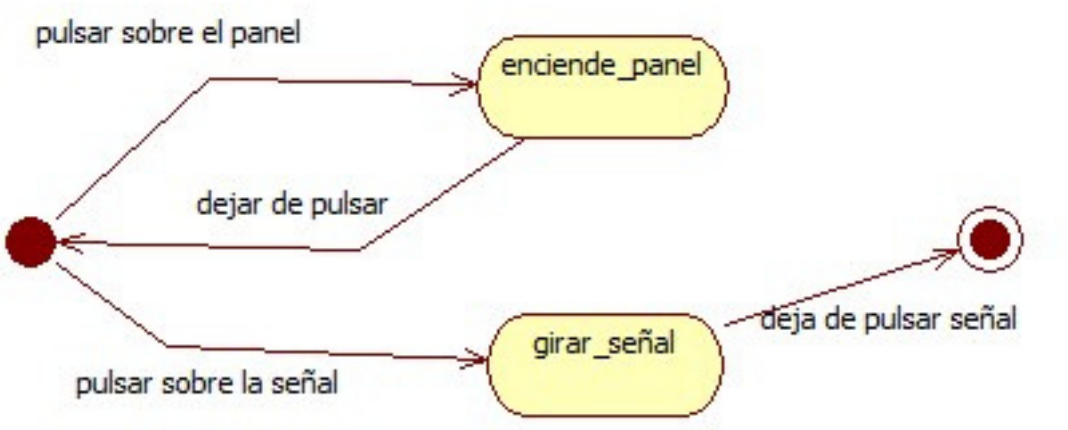
Esta escena corresponde al nivel más complicado de nuestra aplicación, y consiste en dar con el boomerang a todos los palos o cilindros del giratorio. Cuando un objeto cilindro es golpeado por el boomerang, el boomerang se destruye y el cilindro cambia de textura, para que podamos observar que ese cilindro ya está dado.

El estado jugando, como en las anteriores escenas, se produce cuando ya están todas variables y los objetos inicializados y listos para comenzar el juego. La escena posee una base cilíndrica que gira constantemente durante la partida, y con 10 cilindros sobre ella (objetivos).

Los botones de ayuda y salir son los mismos que en las anteriores escenas, con la diferencia que tienen como resultado otro objeto prefabs, ya que cuando es presionado el botón este identifica en que escena se encuentra y dependiendo de cuál sea crea un objeto u otro. De este modo podemos compartir objetos y códigos en las diferentes pantallas adecuando cada menú a su escena.

Cuando se consigue el objetivo, se instancia o se crea un objeto con una textura 2D informativa de que se ha conseguido el objetivo, pero ya solo te da opción a ir a menú ya que es el último nivel.

Diagrama de estado de la escena de créditos "créditos":



Esta es una escena informativa en la cual hemos utilizado el panel para dar la información del creador del proyecto y hacerlo de este modo más atractivo.

En la escena hay una señal con el mensaje volver, es utilizada obviamente para volver al menú, y al pulsar sobre ella, mientras se mantenga el dedo encima, la señal gira hasta que se deje de pulsar y se cargue la escena de menú.

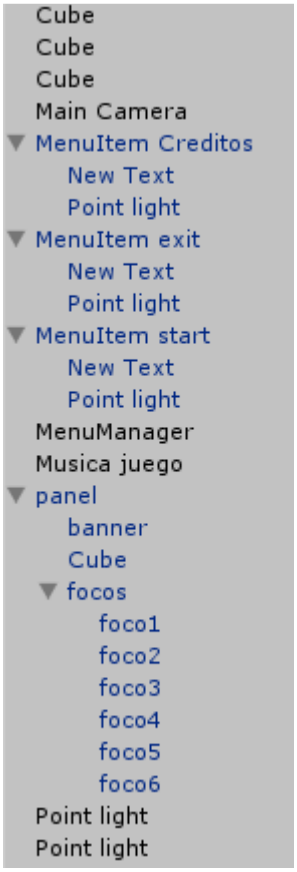
3.4.Diseño e Implementación:

En este apartado vamos a ir explicando cómo hemos realizado la implementación de nuestro proyecto por escenas.

Escena del Menú:

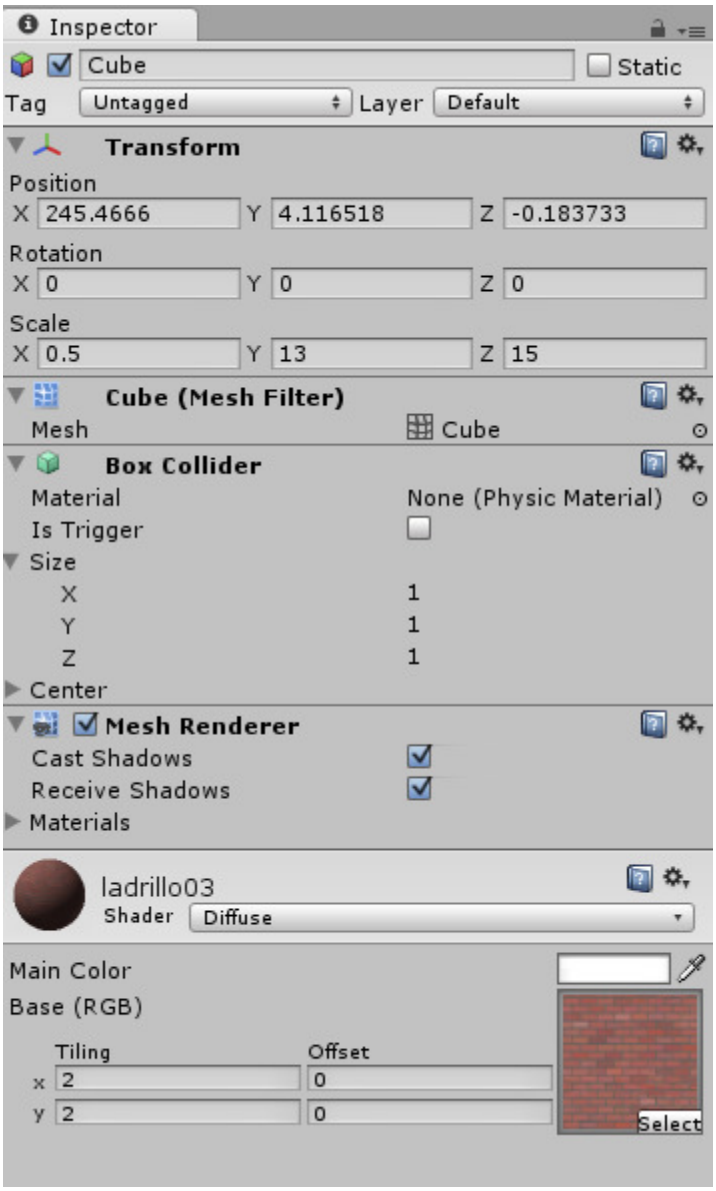
Tanto la escena del menú y la escena del menú dos se utilizan prácticamente los mismos objetos con diversas modificaciones que también explicaremos a continuación.

Veamos el esquema de objetos inicial de la escena menú “esc_menu”:

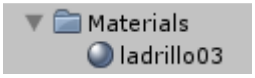


En esta imagen aparecen todos los objetos, ordenados alfabéticamente, que se encuentran en escena. Los objetos pueden destruirse y desaparecer de la lista, o deshabilitarse, en este caso el nombre del objeto tomara un tono gris. Los objetos de azul de la lista son objetos que están prefabricados y que han sido añadidos a la escena. Esto resulta útil cuando vas a utilizar los mismos objetos en diferentes escenas, para no tener que crear en cada escena un objeto nuevo.

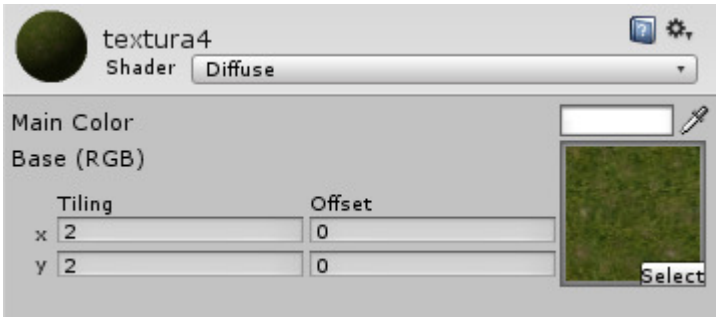
Los tres primeros objetos “Cube” corresponden a las dos paredes de ladrillo y al suelo. En las dos paredes de ladrillo se ha aplicado la misma textura.



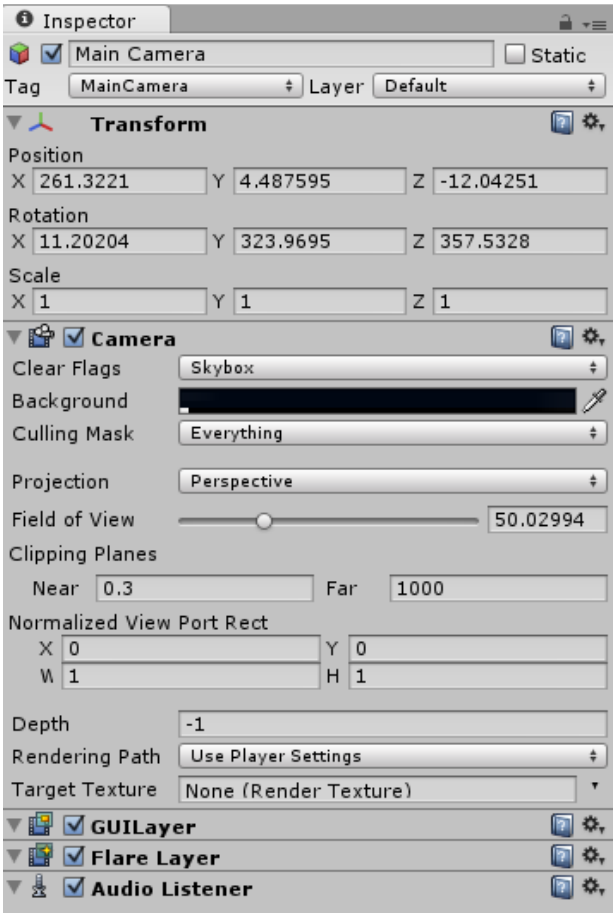
Cuando tienes seleccionado un objeto en la lista de la vista **Hierarchy**, aparecen todos los componentes que contiene el objeto en la vista **Inspector**. Como podemos observar en esta imagen, este es la vista Inspector del objeto Cube. Posee el componente **Transform** con el cual puedes controlar la posición y rotación del objeto en escena. **Box Collider** es utilizado para detectar colisiones pero en esta escena este componente no será relevante ya que no vamos a controlar ninguna colisión de objetos contra la pared. Como se puede observar en el componente **Mesh Renderer,** la pared está configurada para recibir sombras y tiene asignada la textura “ladrillo”. Cuando se asigna una textura a un objeto, se crea un directorio en la carpeta raíz de la escena con los materiales de los objetos y se crea un material con la textura asignada.



Para la creación de las paredes y el suelo hemos utilizados los objetos propios de Unity. Mediante la barra de herramientas si seleccionas **GameObject->Create Other->Cube** crea un cubo que puede ser modificado a través del componente **Transform** de la vista Inspector o por medio de un script para obtener el cubo deseado. El Cube creado para el suelo contiene otra textura y por lo tanto otro material.



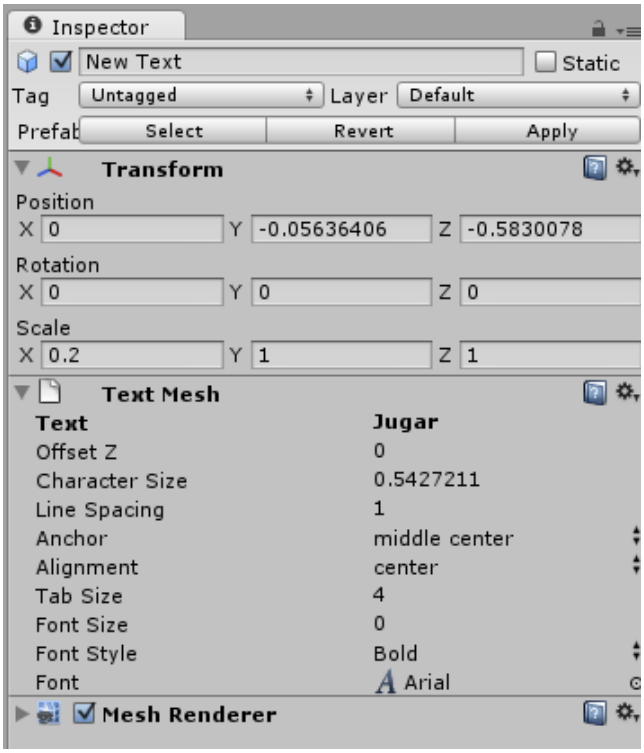
El siguiente objeto de la lista es **Main Camera**, este el encargado de mostrar la escena. Por defecto cuando se crea una escena nueva siempre se crea una camera (GameObject->Create Other->Camera).



Con el componente **Transform** controlamos la posición de la cámara y por consiguiente el ángulo y la perspectiva que queremos ver en nuestro juego. En el componente **Camera** se puede configurar varios atributos de la cámara, para obtener diferentes resultados. En mi aplicación el valor que he modificado principalmente es "Field of View" con el que configuras la distancia focal, de este modo obtienes que los objetos se vean más o menos lejos a pesar de que se encuentren en la misma posición. **Main Camera** con el componente **GUILayer** permite introducir a la imagen resultante de la escena componentes de la interfaz de Unity. Y otro importante es **Audio Listener** que es el receptor de sonidos de la escena, a través de este componente los sonidos que producen otros objetos serán escuchados. El sonido de Unity está configurado tridimensionalmente, con esto quiero decir que dependiendo donde se encuentre el emisor de sonido(Audio Emitter) y el receptor (Audio Listener), el sonido se escuchará más o menos según la distancia.

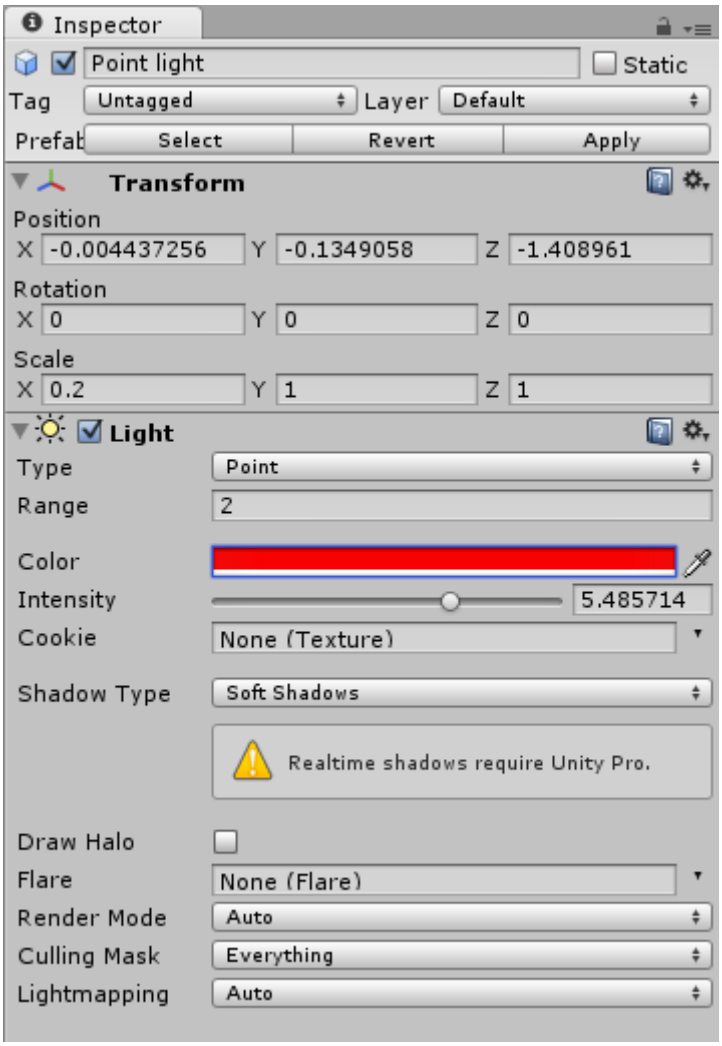
Los tres elementos de la lista siguientes son los objetos que componen el menú. Tanto "**MenuItem Creditos**", "**MenuItem exit**" como "**MenuItem start**" son instancias del mismo Prefab (objeto prefabricado).

Esta imagen corresponde al Prefab de la vista Project. Como se observa en la imagen, el elemento del menú está compuesto por un objeto padre (MenuItem) y dos hijos (New Text y Point light).



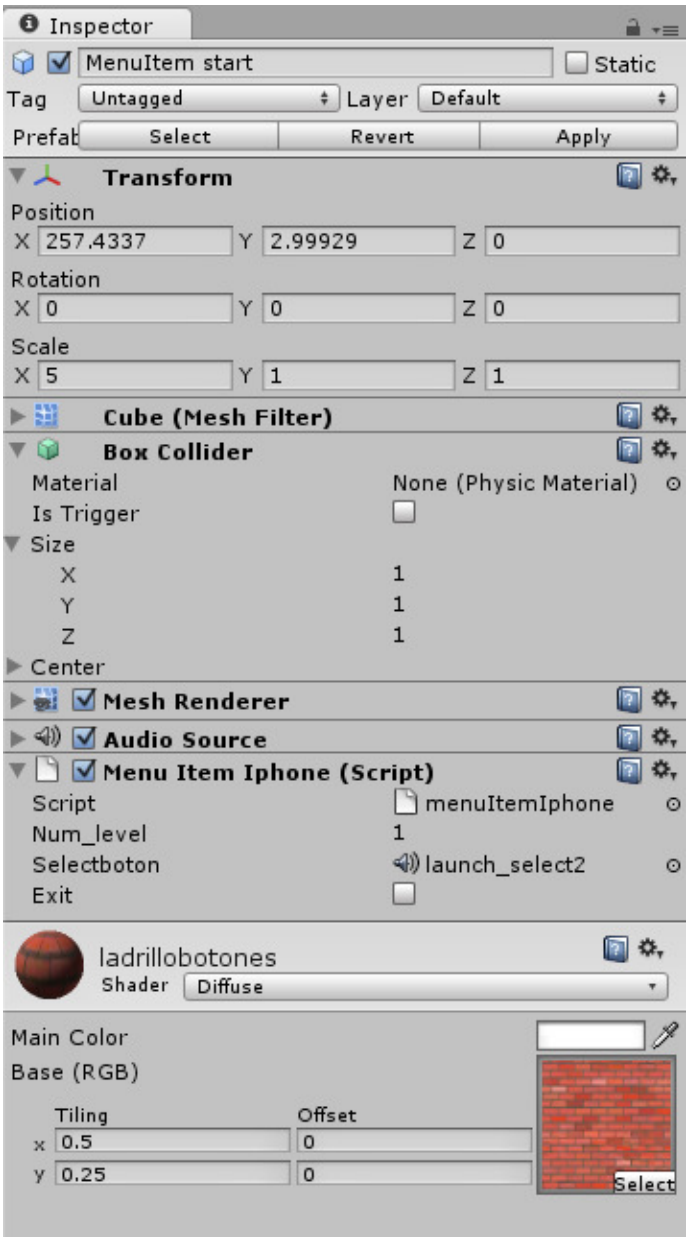
En cuanto a los hijos, New Text es el encargado de mostrar el texto del botón del menú. Cada New Text de cada MenuItem en el menú tiene un texto diferente (Jugar, Creditos y Salir). A pesar de que los tres elementos del menú provengan del mismo objeto prefabricado, se pueden modificar diversos valores para obtener diferentes resultados.

Cada elemento del menú también posee un punto de luz propio (Point Light) para que el mensaje del botón se vea claramente.



El componente Light del objeto es de tipo Point como se puede observar en la imagen. Con el rango configuramos la zona afectada por ese punto de luz y con Intensity la intensidad. Modifique el color de la luz para que no se “quemara” (cuando un objeto no se ve definido por causa de una fuerte luz) el objeto y elegimos el color rojo que se asemeja más al color del ladrillo.

Si seleccionamos el objeto **MenuItem** en la vista **Inspector** se puede observar los componentes de este objeto como se ve en la siguiente imagen.



Vamos a hacer referencia a los componentes de este objeto que interactúan en la escena. **Box Collider** recibe las colisiones de otros objetos, como por ejemplo el toque de un dedo. Este es el evento que queremos controlar. Cuando el usuario toca con el dedo el botón el objeto modifica su posición un poco hacia delante resaltando así el objeto que tenemos seleccionado.

El componente **Audio Source** es el encargado de emitir el sonido cuando el botón es seleccionado para que Main Camera con su componente Audio Listener recoja ese sonido.

El objeto, como se ve en la imagen anterior tiene asignado el script “menuItemIphone” que tiene tres variables públicas que se pueden ver en la vista Inspector. Num_level es el número de escena de la aplicación a cargar cuando el botón deje de ser pulsado.

El script es el siguiente:

```
using UnityEngine;
using System.Collections;

public class menuItemIphone: MonoBehaviour {
    public int num_level;
    public AudioClip selectboton;
    public bool exit;
    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
    void FingerBegin (iPhoneTouch evt) {
        audio.PlayOneShot(selectboton);
        transform.Translate(Vector3.forward *-1);
    }

    void FingerMove (iPhoneTouch evt) {
        //Debug.Log("Paseando");
    }

    void FingerEnd (iPhoneTouch evt) {
        transform.Translate(Vector3.forward *1);
        if(exit)
            Application.Quit();
        else
            Application.LoadLevel(num_level);
    }

    void FingerCancel (iPhoneTouch evt) {
        transform.Translate(Vector3.forward *1);
        if(exit)
            Application.Quit();
        else
            Application.LoadLevel(num_level);
    }
}
```

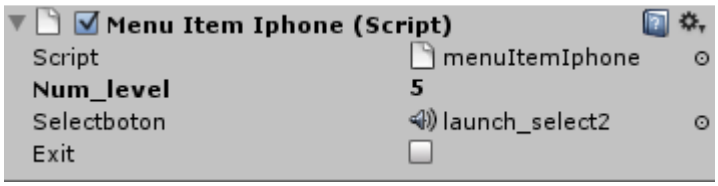
Tiene cuatro funciones con diferentes estados del Touch (toque con el dedo). En FingerBegin configuramos que se reproduzca el sonido del botón seleccionado cuando comienza a ser tocado y modificamos su posición para resaltar el botón. Y en los métodos FingerEnd y FingerCancel reseteamos la posición inicial que tenía el botón y si

no es el botón de salir (exit==true) entonces cargamos el nivel que tenga asignado el botón. Cada botón del menú tendrá un número de nivel diferente.

Estas funciones son llamadas por Menu Manager que es el encargado de gestionar las colisiones de los Touch en los objetos.

Los tres elementos del menú tienen el mismo script asignado, con diferentes valores de nivel a cargar. Este valor se asigna mediante la variable pública “num_level” en el inspector.

Si observamos la siguiente imagen vemos que el valor de la variable es diferente al del anterior ejemplo.



El componente del elemento “Creditos” tiene el 5 debido a que en Build Settings la escena de los créditos tiene asignada la numeración 5. La variable aparece en negrita en el Inspector cuando el valor es diferente al valor por defecto del Prefab.

El siguiente elemento de la lista “**MenuManager**” es el encargado de controlar todos los eventos que sucedan en la escena con respecto al menú. Para controlar estos eventos asignamos el script “FingerManager.cs” a un objeto vacío (empty) y, de este modo, enviar señales a otros objetos sobre si están siendo tocados con el dedo o cualquier otro evento que interese.

```
using UnityEngine;
using System.Collections;

public class Finger {
  public iPhoneTouch touch;
  public bool moved = false;
  public ArrayList colliders = new ArrayList();
}

public class FingerManager : MonoBehaviour {

  public ArrayList fingers = new ArrayList();

  void Update () {
    RaycastHit[] hits;
    foreach (iPhoneTouch evt in iPhoneInput.touches) {
      if (evt.phase==iPhoneTouchPhase.Began) {
        Finger finger = new Finger();
        finger.touch = evt;
        Ray ray = Camera.main.ScreenPointToRay(evt.position);
        hits = Physics.RaycastAll(ray);
        foreach (RaycastHit hit in hits) {
          finger.colliders.Add(hit.collider);
          GameObject to = hit.collider.gameObject;
          to.SendMessage("FingerBegin",evt,SendMessageOptions.DontRequireReceiver);
        }
        fingers.Add(finger);
      }
      else if (evt.phase==iPhoneTouchPhase.Moved) {
        for (int i=0;i<fingers.Count;++i) {
          Finger finger = (Finger)fingers[i];
          if (finger.touch.fingerId==evt.fingerId) {
            finger.moved = true;
            foreach (Collider collider in finger.colliders) {
              if (collider==null) {continue;}
              GameObject to = collider.gameObject;
              to.SendMessage("FingerMove",evt,SendMessageOptions.DontRequireReceiver);
            }
          }
        }
      }
```

```
        else if (evt.phase==iPhoneTouchPhase.Ended || evt.phase==iPhoneTouchPhase.Canceled) {
            Ray ray = Camera.main.ScreenPointToRay(evt.position);
            hits = Physics.RaycastAll(ray);
            for (int i=0;i<fingers.Count;) {
                Finger finger = (Finger)fingers[i];
                if (finger.touch.fingerId==evt.fingerId) {
                    foreach (Collider collider in finger.colliders) {
                        if (collider==null) {continue;}
                        bool canceled = true;
                        foreach (RaycastHit hit in hits) {
                            if (hit.collider==collider) {
                                canceled = false;
                                GameObject to = collider.gameObject;
                                to.SendMessage("FingerEnd",evt,SendMessageOptions.DontRequireReceiver);
                            }
                        }
                        if (canceled) {
                            GameObject to = collider.gameObject;
                            to.SendMessage("FingerCancel",evt,SendMessageOptions.DontRequireReceiver);
                        }
                    }
                    fingers[i] = fingers[fingers.Count-1];
                    fingers.RemoveAt(fingers.Count-1);
                }
                else {
                    ++i;
                }
            }
        }
    }
}
```

En este script tenemos por un lado la declaración de la clase Finger donde almacenará información sobre el toque (touch), si el dedo se está moviendo con un booleano y una lista para guardar las colisiones con objetos de las escenas.

Por cada toque sobre la pantalla se crea un evento que es analizado y según el estado (phase) en la que se encuentre se realizarán unas instrucciones u otras.

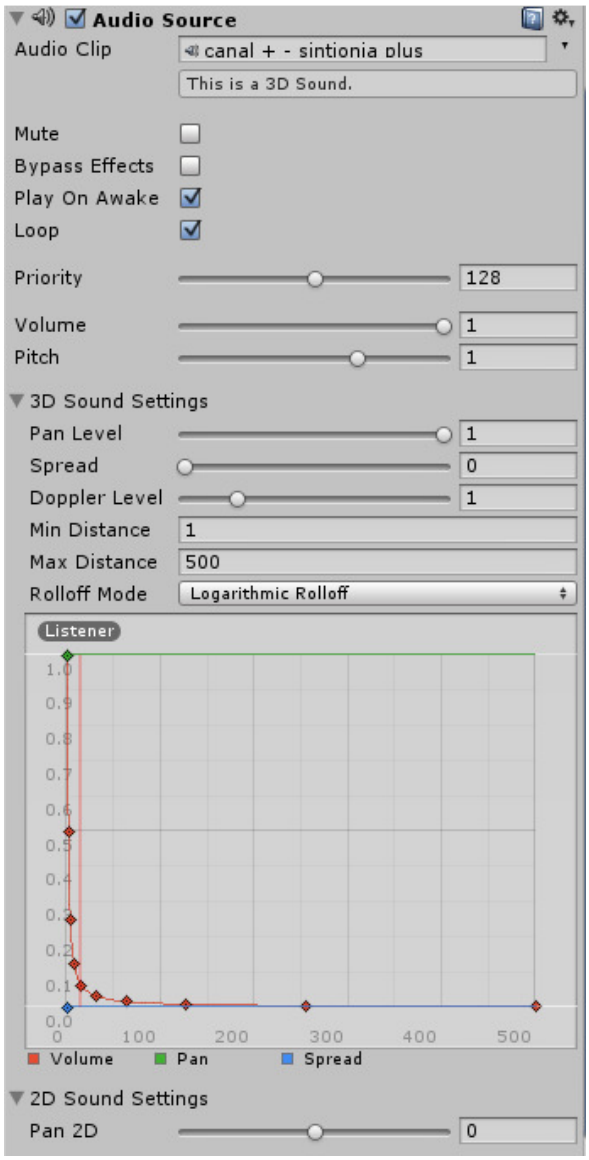
En cuanto al estado en el cual el toque acaba de suceder o comenzar, se identifica el objeto sobre el cual se está efectuando el toque, se añade ese objeto al "Finger" mediante la instrucción *"finger.colliders.Add(hit.collider);"* y se envía una señal al objeto pulsado o tocado con la instrucción "*to.SendMessage("FingerBegin",evt,SendMessageOptions.DontRequireReceiver);"*.De este modo, el objeto recibirá la señal y ejecutará el método "FingerBegin".

Además se añade el nuevo Finger al **ArrayList fingers**, para que en la fase "Moved" busque si el toque o finger ya se encuentra en la lista y asi poder poner a TRUE o

activar el booleano que indica que el dedo (finger) se está moviendo por la pantalla. También, se enviará un mensaje al objeto que esté recibiendo la colisión y éste ejecute el método “FingerMoved”.

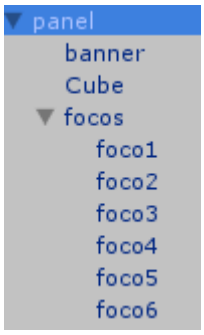
En cuanto a las fases de terminar el toque (iPhoneTouchPhase.Ended) y cancelado (iPhoneTouchPhase.Canceled), lo que realiza el script es un bucle sobre todos los fingers para identificar el actual en la lista, y así, recorrer toda la lista de objetos que tiene el “Finger” para así enviar el mensaje al objeto que estaba siendo pulsado de que ya ha terminado de ser pulsado y ejecute los métodos “FingerEnd” y “FingerCancel” y luego eliminar el Finger de la lista “fingers” con *“fingers.RemoveAt(fingers.Count-1);”*.

El siguiente objeto de la lista es “**Musica juego**”, un objeto vacío (empty), en la cual se le ha asignado un componente de la clase **AudioSource** para que reproduzca una música de fondo en el menú.



Tenemos seleccionada la pista de audio en Audio Clip y las opciones de **PlayOnAwake**, para que empiece a reproducir la pista en el método Awake, y que realice un bucle seleccionando la opción **Loop**. Luego el componente tiene otra serie de propiedades para controlar todos los aspectos del sonido 3D y 2D.

El objeto Panel tiene varios sub-objetos y a su vez otros sub-objetos. Esta es la jerarquía de este objeto:



Panel es el encargado de ejecutar las instrucciones necesarias para que cuando el panel este pulsado se ilumine y cuando el panel no esté siendo pulsado no se ilumine.

Para ello, panel tiene asignado un script llamado "script_panel.cs".

Este script tiene una variable pública de tipo "GameObject" para asignar a través del interfaz la variable y no tener que introducir código en nuestro script, es una de las facilidades que otorga Unity.

La variable asignada va a ser un objeto al cual se le notificará cuando debe ejecutar el método "apaga_focos".

Veamos el script:

```
using UnityEngine;
using System.Collections;

public class script_panel : MonoBehaviour {
    public GameObject focos_not;
    // Use this for initialization
    void Start () {
        focos_not.SendMessage("apaga_focos",SendMessageOptions.DontRequireReceiver);
    }

    // Update is called once per frame
    void Update () {

    }
    void FingerBegin (iPhoneTouch evt) {
        gameObject.SetActiveRecursively(true);
    }

    void FingerMove (iPhoneTouch evt) {
        gameObject.SetActiveRecursively(true);
    }

    void FingerEnd (iPhoneTouch evt) {
        focos_not.SendMessage("apaga_focos",SendMessageOptions.DontRequireReceiver);
    }

    void FingerCancel (iPhoneTouch evt) {
        focos_not.SendMessage("apaga_focos",SendMessageOptions.DontRequireReceiver);
    }
}
```

Como podemos ver, en el método Start, nada más instanciarse el objeto, apaga los focos para que así de primeras los focos estén apagados. Luego tenemos los métodos que se ejecutan según la fase del toque, de tal modo que cuando el dedo comience a tocar sobre el panel o se esté moviendo se activen los focos y cuando deje de ser pulsado se envíe un mensaje al notificador de los focos para que desactive los focos. Para hacer que los focos se apaguen el método "apaga_focos" tiene la instrucción *"gameObject.SetActiveRecursively(false);"* que gracias a la jerarquía desactiva todos los hijos del objeto que ejecuta esta instrucción y se desactiva a sí mismo también. Así pues, cuando el toque (touch) acaba de comenzar o se está moviendo, con la instrucción *"gameObject.SetActiveRecursively(true);"* ejecutada desde "panel" activa los focos (puntos de luz).

El objeto del panel es descargado de internet, y le hemos añadido una cubo para introducir la textura con nuestro logo del juego. A parte, hemos creado el notificador de focos (focos) y hemos añadido unos puntos de luz justo donde el modelo 3D tenía los focos pero no iluminaba.

Los dos siguientes objetos corresponden a la iluminación de la escena. Son dos puntos de luz colocados de tal manera que ilumine de la manera deseada la escena. Para crearlos hemos seleccionado en el menú GameObject->Create Other->Point Light. Y para conseguir los efectos deseados hemos variado los valores de los atributos del objeto, tales como la intensidad y el rango de la luz.

Veamos ahora la lista de objetos de la vista Hierarchy de la escena del menú dos "**esc_menu2**".

Como se puede observar esta escena es prácticamente igual que la anterior con ligeras variaciones.

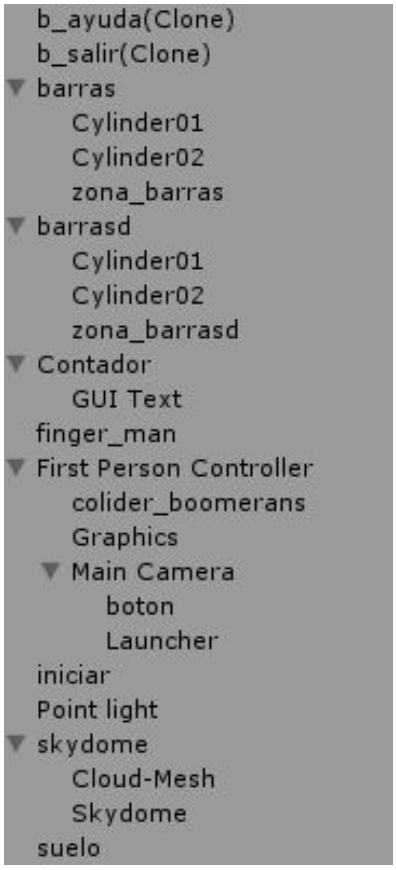
Como en esta parte del menú hay más opciones es necesario tener un elemento más del menú (MenuItem). Los valores de la variable Text del objeto hijo New Text cambia

para dar el para mostrar otros textos. La variable “num_level” de los ítems del menú también cambia para cargar diferentes escenas haciendo referencia cada elemento del menú a una escena específica. Para el funcionamiento correcto del menú y el juego la lista de escenas debe estar configurada siempre de la misma manera para no cambiar el orden y la numeración de las escenas.

Los demás objetos, MenuManager, Musica juego, panel y los puntos de luz, son los mismos que en la escena anterior y poseen el mismo funcionamiento.

Escena Boomerang “escena_barras”:

Veamos el esquema de la vista Hierarchy donde se muestran todos los objetos de la escena justo al comenzar. Durante la partida según las decisiones del usuario aparecerán y desaparecerán otros objetos.

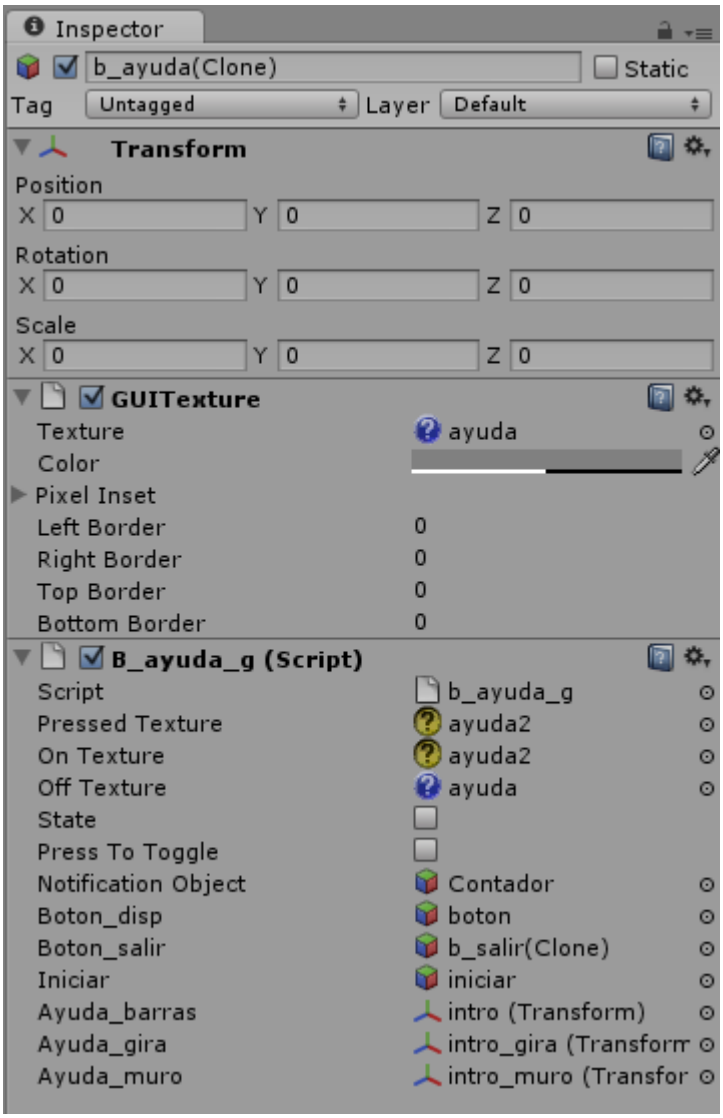


Cuando empieza la pantalla se crean los objetos del botón de ayuda (b_ayuda) y el botón de salir (b_salir). Son dos prefab, que cuando se instancian aparecen en la jerarquía con el sufijo (Clone).

El objeto “b_ayuda(Clone)” es un botón para acceder a las instrucciones de la pantalla o escena en la cual estés. Este objeto se crea en los tres niveles de este juego e identifica en que escena está para mostrar una imagen u otra de instrucciones.

Este objeto consta de una textura 2D la cual cuando es pulsada cambia por otra dando el efecto de botón dinámico.

Ahora veremos todos los componentes y como hemos realizado su configuración para que el objeto cumpla con lo deseado.



Para realizar las acciones hemos construido el script “b_ayuda_g.cs”, en el cual como se ve en la vista Inspector varias variables publicas que vamos a explicar su función ahora.

- **Pressed Texture**: es la textura cuando esta presionada.
- **On Texture**: es la textura cuando el dedo o cursor esta sobre ella.
- **Off Texture:** es la textura cuando no está presionada ni el dedo sobre ella.
- **State:** cuando es true, se activa la textura onTexture.

Las siguientes variables son para notificar y ejecutar diversos métodos de otros objetos en determinados momentos.

En el método Update del script controlamos los toques (touch) que se producen sobre la textura para así cambiar de textura.

```
void Update()
{
    bool didTouch = false;
    // get all the touches, see if one hits me
    int i;
    for (i = 0; i < iPhoneInput.touchCount; i++) {
        iPhoneTouch touch = iPhoneInput.GetTouch(i);
        Vector3 pos = new Vector3(touch.position.x,touch.position.y,0.0f);
        if (guiText != null) {
            if (guiText.HitTest(pos)) {
                didTouch = true;
                this.doTouchDown();
            }
        } else {
            if (guiTexture.HitTest(pos)) {
                this.doTouchDown();
                didTouch = true;
            }
        }
    }
    if (!didTouch && touchDown) {
        doTouchUp();
    }
}
```

Se realiza un bucle sobre todos los eventos sobre iPhoneInput y se obtiene la posición del touch. De este modo con la instrucción HitTest de la clase GUIText comprobamos si la textura ha sido pulsada y llamamos a la función "doTouchDown". Si no llamamos a la función "doTouchUp".

La función doTouchDown cambia la textura siempre y cuando no la habría cambiado antes. Es decir, como esta función es llamada cada frame que la textura está siendo pulsada, controlamos con la variable booleana "touchDown" si ya hemos cambiado la textura.

```
public void doTouchDown()
{
    if (touchDown) return;
    setTexture(pressedTexture);
    touchDown = true;
}
```

En cambio, cuando deja de pulsar la textura se ejecuta la función doTouchUp. Una vez pulsado el botón hay que parar el contador, notificar al objeto iniciar la pausa, ocultar el botón de disparar y quitar el botón de salir (b_salir).

Para esto se utilizan los mensajes entre objetos, para ejecutar métodos desde el objeto propio.

Se llama a la función “parar” del objeto contador para parar el tiempo y después reanudarlo después de salir del menú de ayuda.

Se oculta el botón de disparar para que no se pueda disparar cuando el menú de ayuda está en escena.

Y se quita también el botón de salir para que solo estén accesibles las opciones del menú de ayuda de la escena.

Después, en las siguientes instrucciones, dependiendo de la escena que este cargada se creara un menú de ayuda u otro.

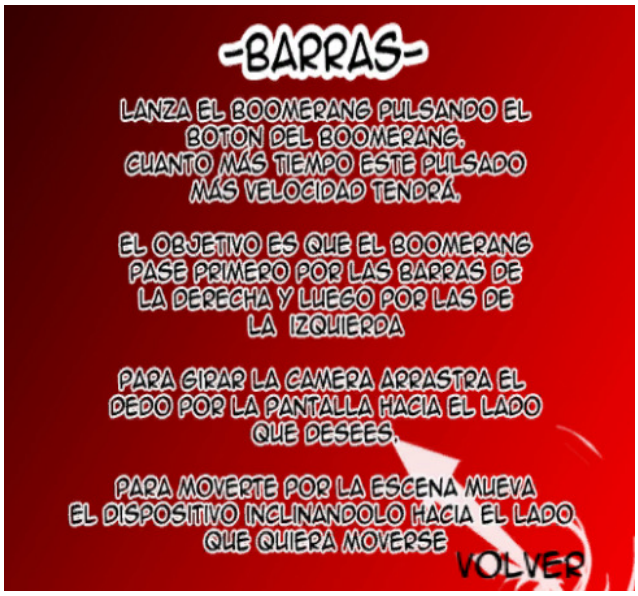
Y por ultimo destruimos el propio botón de ayuda.

```
public void doTouchUp()
{
    if (!touchDown) return;
    if (pressToToggle) state = !state;
    if (state) {
        setTexture(onTexture);
    } else {
        setTexture(offTexture);
    }
    touchDown = false;
    //PARAR CONTADOR, OCULTAR LOS BOTONES DE SALIR Y DISPARAR.
    notificationObject.SendMessage("parar",SendMessageOptions.DontRequireReceiver);
    boton_disp.SendMessage("ocultar",SendMessageOptions.DontRequireReceiver);
    iniciar.SendMessage("pausa_up",SendMessageOptions.DontRequireReceiver);
    boton_salir.SendMessage("quitar_boton",SendMessageOptions.DontRequireReceiver);
    //COMPROBAMOS EN QUE ESCENA NOS ENCONTRAMOS
    //Y CREAMOS EL MENU DE AYUDA
    if(Application.loadedLevel==2){
        Instantiate(ayuda_barras);
    }else if (Application.loadedLevel==3){
        Debug.Log("ajaaaa");
        Instantiate(ayuda_muro);
    }else {
        Instantiate(ayuda_gira);
    }
    //LUEGO DESTRUIMOS EL BOTON DE AYUDA
    Destroy(gameObject);
}
```

Luego el script contiene un método “quitar_boton” con el cual destruye el botón con la instrucción *“Destroy(gameObject);”*. Está destinada para cuando se accione el botón de salir para quitar el botón de ayuda como hemos hecho en nuestro caso con el botón de salir.

Cuando el botón de ayuda es pulsado se crea un objeto del **menú de ayuda** que consiste en una textura con la información de cómo jugar en la escena seleccionada y una textura que construida de forma parecida al botón ayuda para volver a la partida,

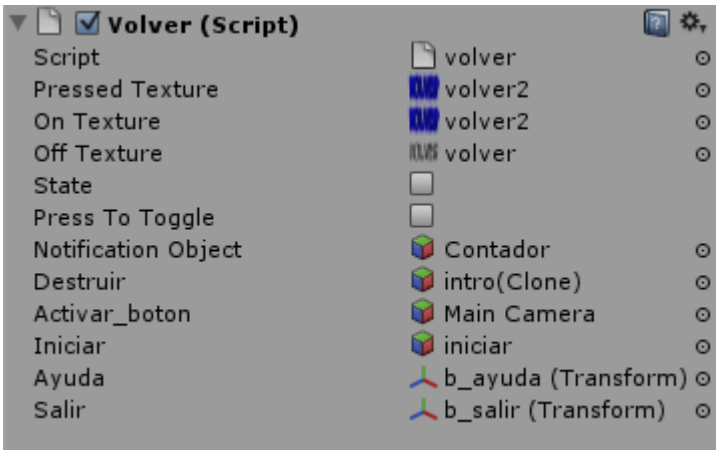
es decir, salir del menú de ayuda y reanudar la partida en el punto que se había parado.



Cuando se crea este objeto aparece el objeto también reflejado en la jerarquía de este modo:

Intro(Clone) contiene una textura que es la imagen con la información y un script que contiene el método **Destruir()** para destruir este objeto y por consiguiente su hijo cuando sea necesario. Este script se asignará a varios objeto los cuales solo vayan a implementar esa funcionalidad.

Volver es un objeto muy parecido al botón de ayuda, contiene dos texturas que se van alternando si las texturas son seleccionadas con el dedo y varias variables para ejecutar varios métodos que hagan posible reanudar la partida.



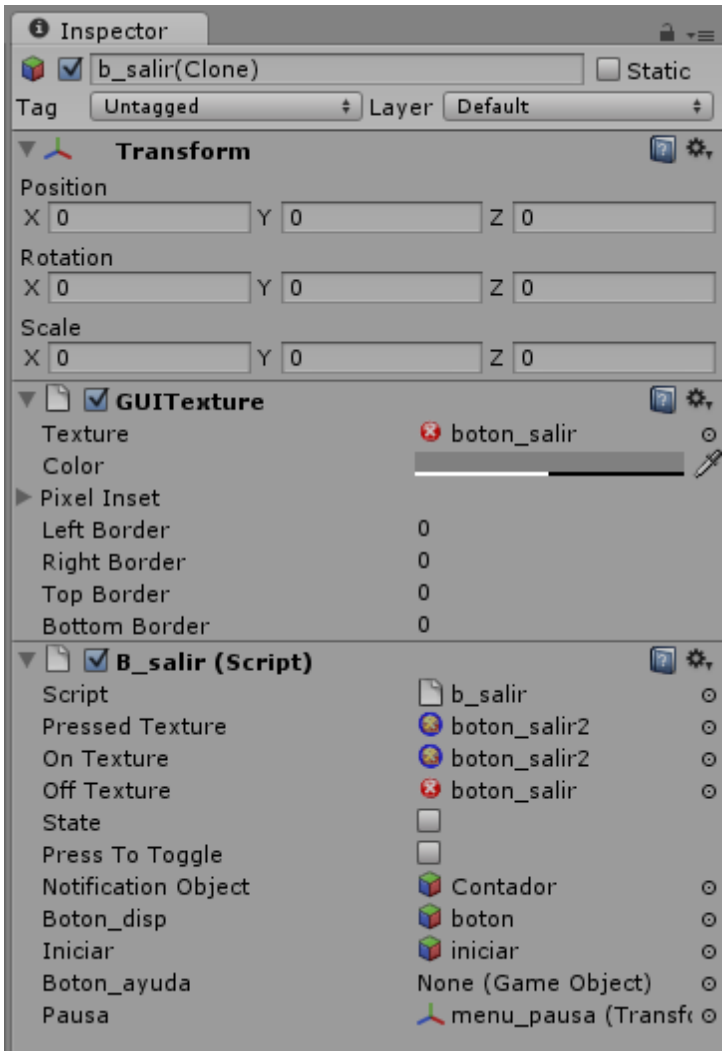
Al dejar de pulsar la textura, como en el botón de ayuda, se ejecuta la función doTouchUp en la cual, en este caso, se ejecutan principalmente las siguientes instrucciones:

```
//HAY QUE ACTIVAR EL BOTON DE DISPARAR
//CONTINUAR EL CONTADOR, CREAR BOTONES DE AYUDA Y SALIR
//Y DESTRUIMOS EL MENU DE INSTRUCCIONES
notificationObject.SendMessage("continuar",SendMessageOptions.DontRequireReceiver);
Instantiate(ayuda);
Instantiate(salir);
activar_boton.SendMessage("Activa_game",SendMessageOptions.DontRequireReceiver);
destruir.SendMessage("Destruir",SendMessageOptions.DontRequireReceiver);
Destroy(gameObject);
```

Con estas instrucciones ejecutamos el método del contador continuar para que continúe la cuenta atrás, se crean de nuevo los botones de ayuda y de salir, se activa el botón de disparar y se destruye el menú de ayuda.

En cuanto al **botón de salir**, se encuentra situado arriba a la derecha, te permite salir de la partida en cualquier momento de ella, o pararla un periodo de tiempo hasta que desees reanudarla.

El botón de salir y ayuda son similares, cambian las texturas y el script pero de código son prácticamente iguales debido a que los dos tiene parecida funcionalidad, el botón de ayuda instancia según la escena un determinado menú con una información y el botón de salir, en cambio, siempre crea el mismo objeto al ser pulsado.



Veamos la vista inspector del objeto b_salir(Clone) o botón de salir.

Como se puede observar en la imagen, en las variables de script tiene asignadas dos tipos de texturas que irán intercambiándose con respecto a los estados que vaya pasando la partida (si se pulsa o no). Además, posee otros objetos para notificar y configurar la escena en caso de ser pulsado.

El contador se tiene que parar al ser pulsado como en el caso del botón ayuda, debemos ocultar el botón de disparar y el botón de ayuda.

Para configurar la posición de la textura en el interfaz utilizamos el siguiente código para colocarlo en la esquina superior derecha.

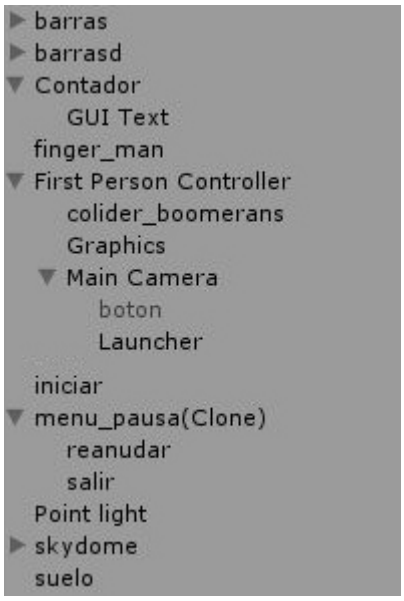
```
void Awake() {
    transform.position = Vector3.zero;
    transform.localScale = Vector3.zero;
    guiTexture.pixelInset = new Rect(Screen.width-40, Screen.height-40,40,40);
}
```

Como se observa en la imagen se utilizan variables de las clases **Transform**, **Vector3** y **GUITexture**.

```
//CREAMOS EL MENU DE PAUSA
Instantiate(pausa);
```

Después de ocultar todos los objetos y parar el contador se instancia o se crea el menú de pausa utilizando la variable asociada a ese Prefab.

Al crear el menú pausa aparece en la lista de objetos Hierarchy el menú de pausa, desaparecen los botones de ayuda y salir, y se deshabilita el botón de disparar. Veamos ahora la lista resultante cuando se instancia el menú de pausa:



Se puede observar que ya no se encuentran en la lista los botones de ayuda y salir, que el botón (hijo de Main Camera) está deshabilitado y que se ha creado el menu_pausa(Clone) y da como resultado la siguiente imagen del juego:

Este es el objeto menú de pausa, tiene una padre con la textura de la imagen que contiene el mensaje "¿qué deseas hacer?" y dos hijos que funcionan como dos

botones construidos también con dos texturas, una para cuando está seleccionada y otra para cuando no, y que tienen como función dar la opción de reanudar la partida en el punto en la cual se paró, o salir al menú.

El botón **reanudar**, está construido de la misma manera que botón de salir y botón de ayuda, se compone de dos texturas las cuales son controladas con un script, en este caso, "m_reanudar.cs".

Como es un objeto instanciado, no se pueden asignar a las variables públicas los objetos de la vista Hierarchy cuando aun no está en la lista, por eso cuando este objeto se instancia, se ejecuta la función Start(), y ahí es donde nosotros asignamos a las variables los objetos con las siguientes instrucciones:

```
notificationObject=GameObject.Find("Contador");
activar_boton=GameObject.Find("Main Camera");
iniciar=GameObject.Find("iniciar");
```

El método Update() es el encargado de controlar los eventos de los toques de pantalla y es el que controla las ejecuciones de los métodos doTouchDown() y doTouchUp(). La parte de código de esta función es la siguiente:

```
void Update()
{
bool didTouch = false;
// get all the touches, see if one hits me
int i;
for (i = 0; i < iPhoneInput.touchCount; i++) {
    iPhoneTouch touch = iPhoneInput.GetTouch(i);
    Vector3 pos = new Vector3(touch.position.x,touch.position.y,0.0f);
    if (guiText != null) {
        if (guiText.HitTest(pos)) {
            didTouch = true;
            this.doTouchDown();
        }
    } else {
        if (guiTexture.HitTest(pos)) {
            this.doTouchDown();
            didTouch = true;
        }
    }
}
if (!didTouch && touchDown) {
    doTouchUp();
}
}
```

La llamada al método doTouchDown() se realiza para cambiar la textura a Pressed Texture, es decir, a la textura cuando está presionada.

El método doTouchUp() tiene como funcionalidad reanudar todos los valores de la partida al punto en el cual la habíamos parado. Esto se consigue con las siguientes instrucciones:

```
//CONTINUA EL CONTADOR, CREAMOS BOTON SALIR Y AYUDA.
notificationObject.SendMessage("continuar",SendMessageOptions.DontRequireReceiver);
Instantiate(salir);
Instantiate(ayuda);
//ACTIVAR EL BOTON DE DISPARAR
//Y DESTRUIMOS EL MENU DE PAUSA
iniciar.SendMessage("pausa_down",SendMessageOptions.DontRequireReceiver);
activar_boton.SendMessage("Activa_game",SendMessageOptions.DontRequireReceiver);
destruir.SendMessage("Destruir",SendMessageOptions.DontRequireReceiver);
Destroy(gameObject);
```

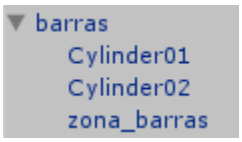
En cuanto a la opción que da el **menú de pausa** de **salir**, la implementación es la misma en cuanto a función Update(), encargada de controlar los evento pero si que hay modificaciones en cuanto a la funcionalidad de este botón cuando es pulsado. Este cambio se ve en las instrucciones del método doTouchUp() de su script “m_salir.cs”.

```
//CARGAMOS EL MENU
Application.LoadLevel(0);
```

Cuando el botón es pulsado se carga el nivel cero, que en nuestra aplicación corresponde con la escena del menú.

Hasta ahora hemos explicado la implementación de los botones de salir, de ayuda, y de los menús resultantes de de haber presionado estos botones. Estos objetos van a ser utilizados en las demás pantallas o escenas del juego por lo que no será necesario volver a explicar su implementación, solamente su diferencias.

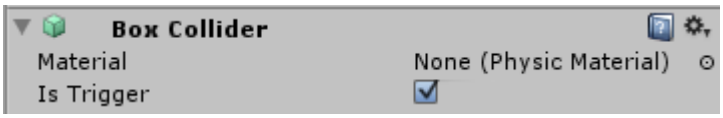
La implementación de las **barras** de la primera pantalla, procede del mismo Prefab pero por motivos de funcionalidad han variado sus scripts y variables.



Las **barras** situadas a la **derecha** de la escena y las barras de la izquierda tienen la misma estructura, dos cilindros y una zona_barras con la cual se detecta si pasa por medio de las dos barras. Lo diferente es que cuando el boomerang pasa por la zona de barras unas barras realizan unas funciones y las otras barras otra función.

En cuanto a las barras de la derecha, para explicar la funcionalidad vamos a explicar la zona de barras la cual es la que controla si el boomerang pasa o no.

El objeto “zona_barras” es un trigger, es decir, un collider configurado como trigger, para deje atravesar los objetos y poder así controlar cuando un objeto pasa por la zona.



Las funciones que controlan las colisiones sobre trigger son OnTriggerEnter(), OnTriggerExit() y OnTriggerStay(), pero en nuestro caso utilizaremos OnTriggerEnter(), la cual hemos implementando en el script asociado a la zona de las barras de la derecha “zona.js”.

```
static var bol : boolean;
var luzverde : Transform;
function Update () {
}
function OnTriggerEnter(coll : Collider) {
    if (coll.tag=="boo"){
                bol=true;
                Instantiate(luzverde,transform.position,transform.rotation);
        }
}
```

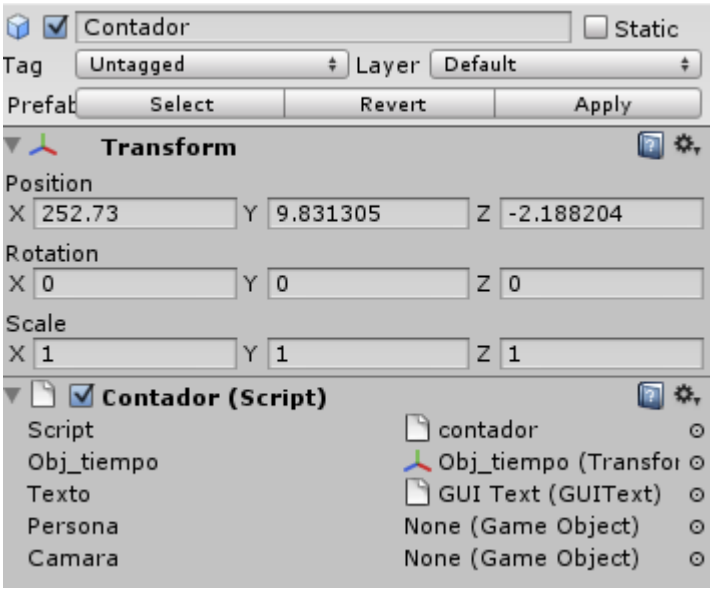
En este script controlamos si el objeto “collider” está etiquetado como “boo”, es decir, es el boomerang, y, en caso correcto, activamos la variable bol a true y instanciamos un objeto para emitir unas partículas para indicar que el boomerang ha pasado la zona de barras.

En cuanto a las **barras** de la **izquierda**, segundo objetivo, tiene un script asociado a la zona de barras de la izquierda con el cual comprueba si ya ha pasado por la zona de barras de la derecha para así incrementar en 1 el contador de la partida.

```
var obj_conseguido : Transform;
var luzverde : Transform;
var notificationObject : GameObject;
var inicio : GameObject;
var boton_fire : GameObject;
var boton_salir : GameObject;
static var cont : int;
function Start() {
    //para que al pasar de escena comience de cero
    cont=0;
}
function Update () {
}
function OnTriggerEnter(coll : Collider) {
    if ((coll.tag=="boo")&&(zona.bol==true)){
                Debug.Log("CONSEGUIDO");
                Instantiate(luzverde,transform.position,transform.rotation);
                cont = cont + 1;
                Debug.Log(cont);
                if (cont == 3) {
                    Instantiate(obj_conseguido);
                    //Launcher.fin_juego=true;
                    //MouseLook.cons=true;
                    notificationObject=GameObject.Find("Contador");
                    boton_salir=GameObject.Find("b_salir(Clone)");
                    notificationObject.SendMessage("parar",SendMessageOptions.DontRequireReceiver);
                    inicio.SendMessage("pausa_up",SendMessageOptions.DontRequireReceiver);
                    boton_fire.SendMessage("ocultar",SendMessageOptions.DontRequireReceiver);
                    boton_salir.SendMessage("quitar_boton",SendMessageOptions.DontRequireReceiver);
                }
        }
}
```

En el script, en el método OnTriggerEnter() se comprueba con una condicional si el objeto colisionador es el boomerang (etiqueta “boo”) y si antes ha pasado la zona de las barras de la derecha accediendo a una variable static del script “zona.js”. Si se cumple la condición, se instancia las partículas verdes indicando que ha pasado el boomerang, se incrementa el contandor, y en caso de que el contador sea igual a 3, se habría conseguido el objetivo y se instancia un menú indicando que se ha conseguido, con dos opciones de salir al menú y para acceder al siguiente nivel. Luego se para el contador, se oculta el botón de disparar y de salir.

El siguiente objeto de la lista es el **contador**, que realiza una cuenta atrás de 60 segundos de tiempo para realizar la partida.



El contador es un objeto padre que tiene un GUIText hijo asociado para mostrar el contador por pantalla. Veamos su estructura:

Contador tiene un script asociado en el cual se realizan las operaciones del reloj, y se comprueba si el tiempo se ha agotado para instanciar el objeto de tiempo agotado, y también posee varias funciones para parar un controlar la reproducción del contador.

Veamos ahora como realiza estas funciones el script:

```
function Start () {
    persona=GameObject.Find("First Person Controller");
    camara=GameObject.Find("Main Camera");
}
```

En la función Start asociamos a las variables los objetos First Person Controller y Main Camera, para que cuando se agote el tiempo al mover el dispositivo iPad o arrastrar un dedo sobre la pantalla, la cámara ni el personaje se mueva.

```
function Update()
{
    if (startCounting)
    {
        if(texto.enabled !=true)
            texto.enabled=true;
        if (countStartTime == 0.0) countStartTime = Time.time;
        //contador = 60 - (Time.time - countStartTime);
        //VALOR_INI POR DEFECTO ES 60
        contador = valor_ini - (Time.time - countStartTime);

        if (contador <= 0)
        {
            startCounting=false;
            //PARAMOS EL PERSONAJE
            persona.SendMessage("Nomover",SendMessageOptions.DontRequireReceiver);
            camara.SendMessage("Nogirar",SendMessageOptions.DontRequireReceiver);
            Instantiate(obj_tiempo);
        }
        tiempo= contador;
    }
}
```

Con la variable startCounting se controla que se siga decrementando el contador, luego se habilita el texto si no está habilitado para que se muestre por pantalla. Para realizar el contador inicializamos la variable valor_ini a 60 y utilizamos la clase Time para realizar las operaciones necesarias.

En el caso de que la variable contador llegue a 0, se para el contador, inmovilizamos la persona y la cámara, y creamos el objeto que indica que se ha acabado el tiempo.

```
function iniciar() {
    contador=60.0F;
    startCounting=true;
    persona=GameObject.Find("First Person Controller");
    persona.SendMessage("Simover",SendMessageOptions.DontRequireReceiver);
    //camara.SendMessage("Sigirar",SendMessageOptions.DontRequireReceiver);
}
```

El método iniciar() inicializa la variable contador a 60 segundos, activa la variable booleana startCounting e identifica al jugador para que active su movimiento.

```
function continuar() {
    valor_ini=salva;
    persona.SendMessage("Simover",SendMessageOptions.DontRequireReceiver);
    camara.SendMessage("Sigirar",SendMessageOptions.DontRequireReceiver);
    startCounting=true;
}
```

Continuar es llamado cuando previamente se ha parado el contador y se precisa continuar con la cuenta atrás. Se asigna al valor inicial el valor salvado al parar el

contador en una variable auxiliar (salva), se activa el movimiento y el giro del usuario, y, por último, se actualiza la variable startCounting a true para que comience de nuevo el contador.

```
function parar() {
    //Debug.Log("se ejecuta parar");
    //AL PARAR GUARDAMOS EL VALOR DEL CONTADOR
    //PARA VOLVERLO A ASIGNAR AL CONTADOR AL CONTINUAR
    salva=contador;
    countStartTime=0.0;
    persona.SendMessage("Nomover",SendMessageOptions.DontRequireReceiver);
    camara.SendMessage("Nogirar",SendMessageOptions.DontRequireReceiver);
    //PARAMOS EL CONTADOR Y DESABILITAMOS EL GUITEXT
    startCounting=false;
    texto.enabled =false;
}
```

Al llamar al método parar, se guarda en una variable auxiliar el valor actual del contador, para luego poder reanudar la cuenta desde ese valor, se resetea la variable countStartTime y paralizamos el movimiento del usuario. Y por ultimo startCounting la inicializamos a false y deshabilitamos el texto para que no se vea el contador cuando el juego está parado.

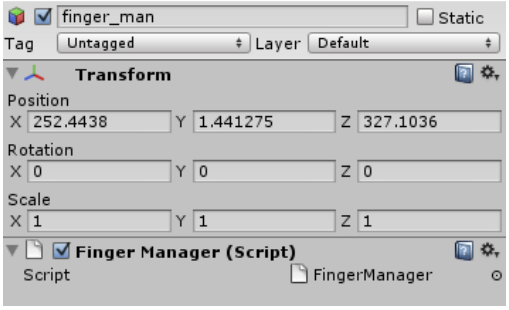
El elemento GUIText que está asociado al contador tiene un script para configurar la forma de mostrar el contador por pantalla. Veamos las instrucciones utilizadas:

```
function OnGUI () {
    //GUI DONDE MOSTRAMOS EL CONTADOR DE TIEMPO
    //var font : Font;
    guiText.material.color=Color.blue;
    guiText.fontStyle = FontStyle.Bold;
    guiText.text = "Tiempo: "+contador.tiempo;
    //guiText.font = font;
    if (Application.loadedLevel==3)
        guiText.pixelOffset = Vector2 (200, 216);
}
```

Modificamos las variables de la clase GUIText como el color de fuente a azul, el estilo de fuente a negrita y el texto a mostrar que es una concatenación de “Tiempo: “ más el valor del contador.

Con pixelOffset controlamos la posición del texto en la pantalla.

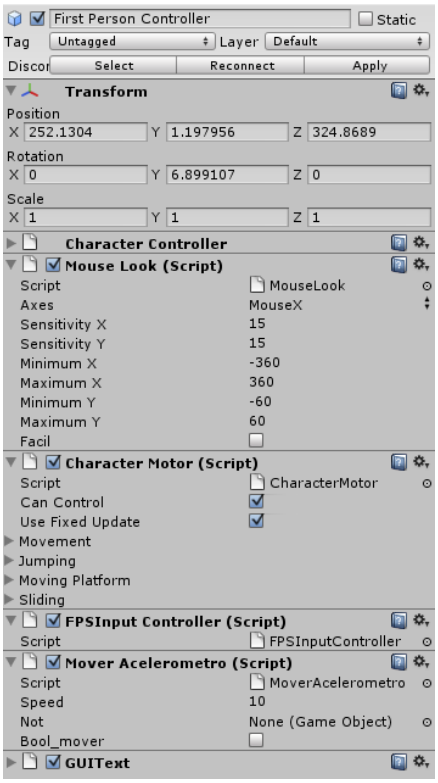
El siguiente objeto de la vista Hierarchy es el “**finger_man**” que se encarga de controlar todos los eventos de la clase Touch. Para ello tiene asociado el script “FingerManager.cs” que envía mensajes a aquellos objetos que están siendo seleccionados en la pantalla con el dedo. Este script ya lo explicamos anteriormente en el apartado de la escena del menú.



El elemento “First Person Controller” es por el cual el usuario interactua con el juego. El un objeto prefabricado que se encuentra en el paquete de Unity “Standard Assets.unitypackage” que se encuentra en el directorio “C:\Program Files\Unity\Editor\Standard Packages”.

Al tener instalado Unity en un PC con sistema operativo Windows 7, no se instaló el apartado de los componentes para iPhone, por lo que al añadir el First Person Controller se añaden muchas funcionalidades para el movimiento con PC y no con iPad o iPhone.

Veamos su vista Inspector:



Como se observa en la imagen el objeto posee muchos componentes que son scripts para controlar el movimiento del ratón (Mouse Look) y el movimiento del personaje.

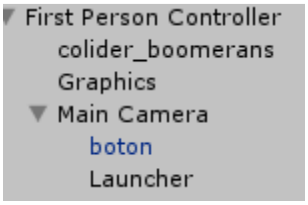
Para adaptar el movimiento y que responda a las señales de entrada del iPad he creado el script “MoverAcelerometro.cs” que ahora explicaremos:

El acelerómetro del dispositivo iPad es el que controla la inclinación del dispositivo en todo momento y manda una señal de entrada con la que nosotros trabajamos.

```
void Update() {
    if (bool_mover) {
        Vector3 dir = Vector3.zero;
        if ((Input.acceleration.y>0.45)||(Input.acceleration.y<-0.45)){
            dir.x = Input.acceleration.y;
        }
        else {
            dir.x =0;
        }
        //si no es este cambiar el otro
        //dir.z = Input.acceleration.x;
        if (dir.sqrMagnitude > 1)
            dir.Normalize();

        dir *= Time.deltaTime;
        transform.Translate(dir * speed);

        //guardamos la posicion del usuario para acceder en otro script
        //con esto eliminamos el boomerang cuando traspase la posicion del usuario
        if (GameObject.Find("boomeran_p(Clone)")){
            not=GameObject.Find("boomeran_p(Clone)");
            not.SendMessage("comp_posicion",transform.position.z);
        }
    }
}
```

La variable “bool_mover” controlar si el personaje se puede mover o no, esto es utilizado cuando se para el juego por los menús de ayuda y salir, por ejemplo. Por razones de optimización pusimos un valor mínimo de inclinación para que comenzara a moverse ya que sino a la mínima inclinación se movía y era muy difícil de controlar. Poner como valor 0,45 equivale más o menos a una inclinación de 45º para un lado o para el otro. Con la función de la clase **Transform** “Translate” se realiza el movimiento teniendo como parámetro la dirección por la velocidad (speed, variable con un valor).

Veamos la jerarquía del objeto del personaje:



El script “MoverAcelerometro.cs” está asignado al objeto “First Person Controller” y con el controlamos el movimiento lateral del personaje. El objeto “colider_boomerans” es un objeto que en principio estaba vacío y le añadimos el componente box collider para detectar las colisiones y así destruir el boomerang en el retorno una vez pasado el personaje. El objeto “Graphics” es el cilindro que se ve en la siguiente imagen, es un objeto que viene en el prefab, no como “colider_boomerans” que se creó por motivos de implementación.

Main Camera es la cámara que reproduce el Game View (vista del juego), como es la única cámara que hay en la escena no hay que gestionar que cámara da las imágenes. Para conseguir girar la cámara al arrastrar el dedo por la pantalla creamos el script “girar.cs” y se lo añadimos al objeto Main Camera.

En la función Start() del script inicializamos la variable booleana a true para permitir el giro. Esta variable será modificada por dos funciones en momentos de pausa en el juego.

```
void Start () {
    Debug.Log("SCRIPT GIRAR CAMARA");
    bool_girar=true;
}
```

En Update(), si esta activado la variable “bool_girar”, con un bucle que recorre todos los touch que sucedan en la pantalla, y por cada touch comprobamos si se encuentra en la fase (phase) de desplazamiento. Si es así, comprobamos para qué lado está realizando el movimiento y aplicamos una rotación sobre la cámara con respecto al lado que se este moviendo.

```
    void Update()
{
    if(bool_girar) {
        // get all the touches, see if one hits me
        int i;
        for (i = 0; i < iPhoneInput.touchCount; i++) {
            iPhoneTouch touch = iPhoneInput.GetTouch(i);
            if (touch.phase == iPhoneTouchPhase.Began) {
                //variable de solo lectura
                //touch.phase=iPhoneTouchPhase.Stationary;
            }
            if (touch.phase == iPhoneTouchPhase.Moved) {
                    if(touch.deltaPosition.x>0) {
                        //he cambiado los valores para que vaya mas despacio
                        transform.Rotate(Vector3.up * Time.deltaTime * 5);
                    } else if (touch.deltaPosition.x<0) {
                        transform.Rotate(Vector3.up * Time.deltaTime * -5);
                    } else {
                        //touch.phase=iPhoneTouchPhase.Stationary;
                    }
            }
        }
    }
}
```

Como para hacer desaparecer el botón lo deshabilitamos, y no podemos acceder a él para habilitarlo, lo que hacemos es habilitar desde su padre todos recursivamente, de este modo se habilita también el botón de disparar, por ello creamos la función siguiente.

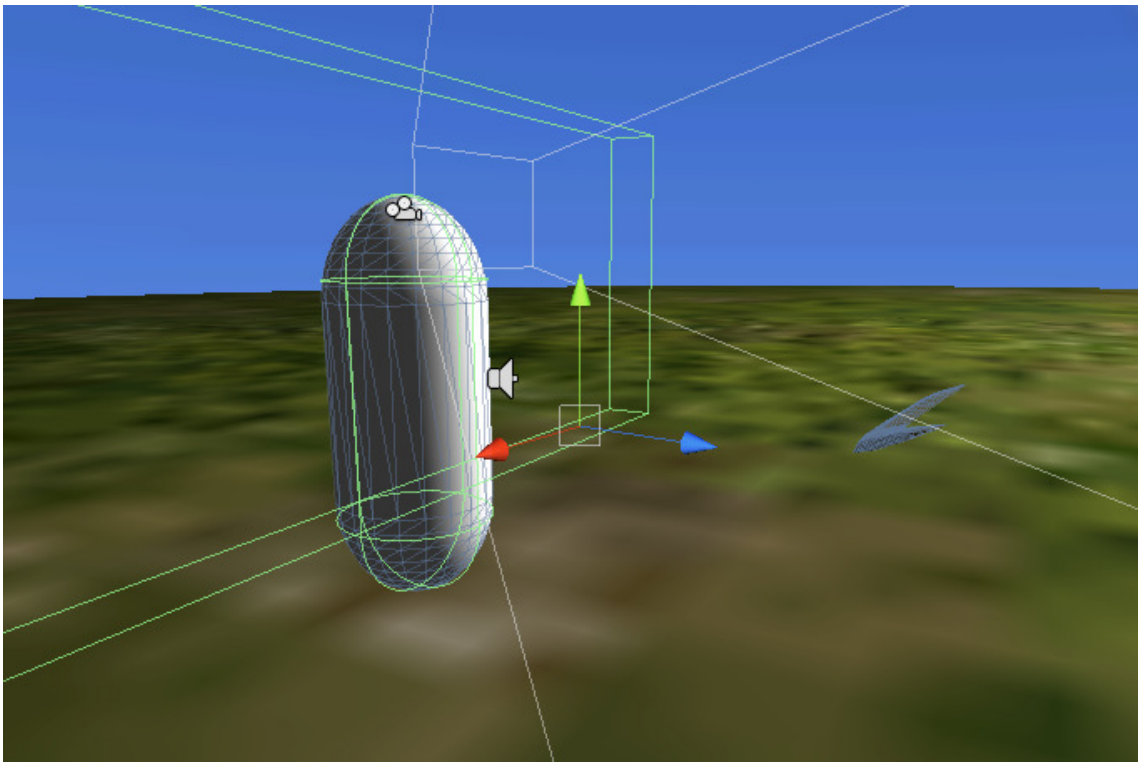
```
void Activa_game(){
    gameObject.SetActiveRecursively(true);
}
```

Las funciones que permiten girar o no la cámara son:

```
void Nogirar() {
    bool_girar=false;
}
void Sigirar() {
    bool_girar=true;
}
```

Si observamos la siguiente imagen, se puede observar la capsula o cilindro que correspondería con el objeto "**Graphics**", la caja de bordes verdes situada detrás de la cápsula que sería el objeto "**colider_boomerans**", se ve también la cámara con un gizmo que es el objeto "**Main Camera**", el gizmo de un altavoz corresponde con el objeto "Launcher", que como produce un sonido al disparar se identifica con el

altavoz, y por último el botón de disparar, un boomerang, que se ve el objeto un poco por delante de la capsula.

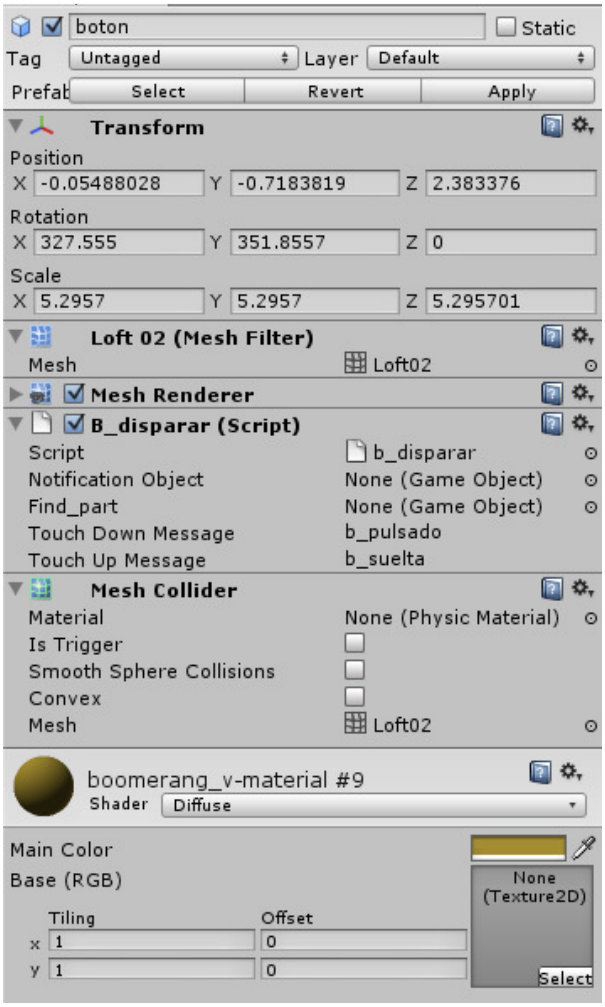


Ahora vamos a explicar dos objetos que añadimos sobre el Prefab por defecto de “First Person Controller” para realizar el lanzamiento del boomerang.

El botón es un objeto que colocamos delante de la cámara en el plano para que realice la función de botón para lanzar el proyectil o boomerang. El modelo 3D se descargó de la siguiente página Web (http://www.turbosquid.com/3d-models/free-max-mode-boomerang-aboriginal-gaming/273053).

Ahora veremos la imagen con los componentes del botón, pero solo hablaremos de los más trascendentales: **Mesh Collider** y el script “**b_disparar.cs**”.

Esta es la vista Inspector del objeto botón:



Se decidió aplicar **Mesh collider** al objeto ya que no es un objeto con una forma muy básica y con la opción Mesh Collider aplicábamos el detector de colisiones solo a la malla. De esta forma el objeto Finger_man, con el script "FingerManager.cs" lanza mensajes a este objeto para que ejecute las funciones del script "b_disparar.cs".

Vamos a explicar el script por partes:

En el método **Start**() buscamos y asignamos el objeto **Launcher** a nuestra variable para notificar cuándo se está presionando el botón y cuándo se suelta.

```
void Start () {
    if (notificationObject == null) notificationObject = gameObject;
    notificationObject=GameObject.Find("Launcher");
}
```

Mientras que en la función **Update**() solo controlamos si está siendo presionado para hacer que gire el botón.

```
void Update () {
    if(presionado)
        transform.Rotate(0,-10,0);
}
```

Las funciones receptoras de los mensajes de FingerManager son las siguientes:

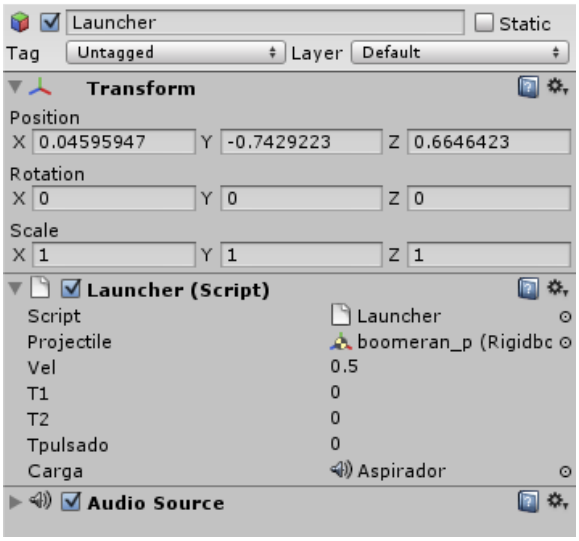
```
//FUNCIONES LLAMADAS POR FINGERMANAGER
void FingerBegin (iPhoneTouch evt) {
    notificationObject.SendMessage(touchDownMessage,SendMessageOptions.DontRequireReceiver);
    presionado=true;
}

void FingerMove (iPhoneTouch evt) {
    presionado=true;
}

void FingerEnd (iPhoneTouch evt) {
    notificationObject.SendMessage(touchUpMessage,true,SendMessageOptions.DontRequireReceiver);
    presionado=false;
    gameObject.active = false;
}

void FingerCancel (iPhoneTouch evt) {
    notificationObject.SendMessage(touchUpMessage,true,SendMessageOptions.DontRequireReceiver);
    presionado=false;
    gameObject.active = false;
}
```

En **FingerBegin**, notificamos al objeto **Launcher** que ha comenzado a presionar el botón y ponemos a **true** la variable presionado para que el botón comience a girar.

En el método **FingerMove** solo reasignamos true sobre la variable booleana "presionado".

Y en las dos funciones restantes, **FingerEnd** y **FingerCancel**, se envía el mensaje a **Launcher** para notificar que se ha dejado de presionar el botón, ponemos "**presionado**" a **false** para que deje de girar el botón y deshabilitamos el objeto botón para que desaparezca el boomerang y de la sensación de que es este el que ha salido disparado

En lo referente al objeto **Launcher**, veamos primero su vista Inspector:



Tiene asociado el script “Launcher.js” y tiene una variable pública “Projectile” que es el objeto que se instanciará y será lanzado cuando el botón sea pulsado.

Cuando el botón llama a la función “b_pulsado”, guardamos en una variable el número de frame en el cual ha sido pulsado y reproducimos el archivo de audio, para dar la sensación de que se está cargando el disparo.

```
function b_pulsado() {
    t1=Time.frameCount;
    Debug.Log("b pulsado");
    Debug.Log(t1);
    audio.PlayOneShot(carga);
}
```

Cuando se deja de pulsar el botón, se manda ejecutar este método. Lo que hace es guardar el frame en el cual ha sido llamada la función, para sacar la diferencia de frames, de esta forma, utilizaremos este valor para controlar la velocidad a la que sale el proyectil. Cuanto mayor es la diferencia mayor tiempo ha sido pulsado el botón, y mayor velocidad recibirá el boomerang.

Este valor se divide por 4 ya que el objeto salía a una velocidad muy elevada.

Se crea el proyectil instanciando el boomerang.

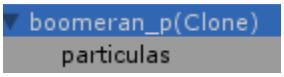
Le aplicamos una fuerza al objeto creado con la función “AddRelativeForce” sobre un vector de velocidad/2 sobre el eje X, 0 sobre el eje Y, y velocidad sobre el eje Z.

Utilizamos
“Physics.IgnoreCollision(instantiatedProjectile.collider,transform.root.collider);”
para ignorar la colisión que pueda tener con su padre (root) y no se destruya el objeto

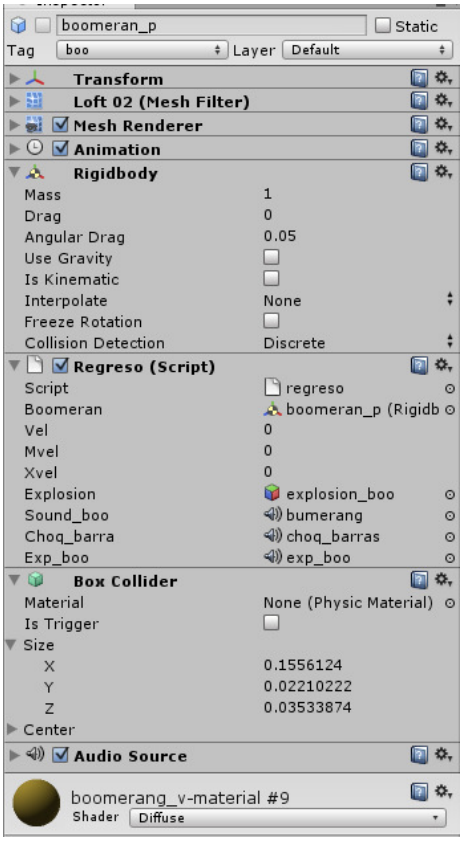
nada más instanciarse. La variable booleana "**bol**" del script "**zona.js**" la inicializamos a false para que en el caso de que en un lanzamiento habría pasado por las primeras barras pero no por las segundas, no contara como correcto el siguiente lanzamiento si solo pasase por las segundas barras. Y por último se para el audio que anteriormente se había activado.

```
function b_suelta(touchdown : boolean ) {
    if (touchdown) {
        Debug.Log("b dispara");
        t2  = Time.frameCount;
        Debug.Log(t2);
        tpulsado= t2-t1 ;
        Debug.Log(tpulsado);

        var instantiatedProjectile : Rigidbody = Instantiate(
                projectile, transform.position, transform.rotation );

        velocidad=tpulsado/4;
        Debug.Log(velocidad);
        instantiatedProjectile.AddRelativeForce(new Vector3(velocidad/2, 0,  velocidad), ForceMode.VelocityChange);
        Physics.IgnoreCollision(instantiatedProjectile.collider,
        transform.root.collider );
        //Ponemos el contador de barras a false
        zona.bol=false;
        audio.Stop();
    }
}
```

Cuando se crea el proyectil se crea un objeto llamado **"boomeran_p(Clone)"** que tiene un hijo llamado partículas que emite un rastro de partículas para ver el recorrido del boomerang.



La vista Inspector de este objeto es la siguiente:



Como se puede observar en la imagen el objeto tiene asignado el script "regreso.js", que es el script encargado de realizar las instrucciones para que el trayecto del proyectil sea el de una elipse aproximadamente.

En la función **Start**() del script se ejecuta nada más crear el objeto por eso reproducimos el audio de lanzamiento de un boomerang, guardamos la velocidad del script "Launcher.js" y realizamos unas operaciones para tener una serie de valores para controlar la trayectoria.

```
function Start() {
    audio.PlayOneShot(sound_boo);
    notificationObject=GameObject.Find("boton");
    vel=Launcher.velocidad;
    xvel=vel/2;
    mvel=vel;
    vel=vel*-2;
}
```

La función Update se ejecuta cada frame, y con los valores obtenidos de la función Start() vamos modificando la fuerza que se aplica al boomerang de manera diferente. Se va decrementando la velocidad mientras "mvel" sea mayor que "vel" para conseguir que el boomerang vuelva, y para el movimiento lateral, mientras "mvel" sea mayor que cero se reduce la fuerza sobre el eje X, y cuando sea menor que 0 y menor que "xvel" se aumenta la fuerza en el eje X.

```
function Update () {
    if (mvel>vel) {
        mvel=mvel-0.2;
        boomeran.AddRelativeForce(new Vector3(0, 0,-0.2), ForceMode.VelocityChange);
        if (mvel>0){
            boomeran.AddRelativeForce(new Vector3(-0.2,0,0), ForceMode.VelocityChange);
        }
        else if(mvel<xvel){
            boomeran.AddRelativeForce(new Vector3(0.2,0,0), ForceMode.VelocityChange);
        }
    }
}
```

Cuando el objeto colisione, registramos la colisión con la función **OnCollisionEnter**(), e identificamos el punto de contacto, e creamos la normal sobre el punto de contacto del objeto para luego instanciar ahí la explosión.

Comprobamos si el choque ha sido con las barras, que están etiquetadas como "b1" y "b2".

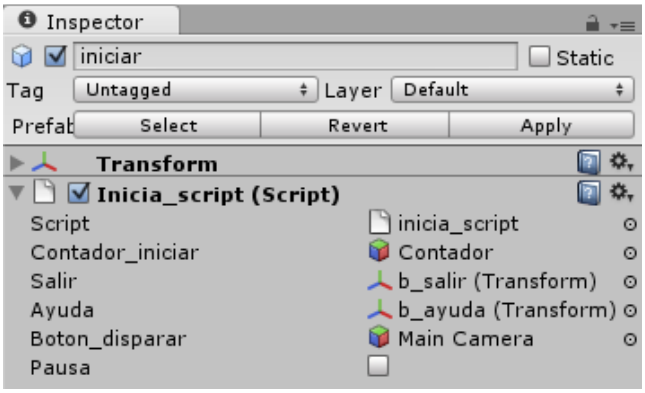
```
function OnCollisionEnter(collisionInfo : Collision){
        var contact : ContactPoint = collisionInfo.contacts[0];
        var rota = Quaternion.FromToRotation( Vector3.up, contact.normal );
        if ((collisionInfo.gameObject.tag=="b2")||(collisionInfo.gameObject.tag=="b1")){
                Debug.Log("choque con barras");
                audio.PlayOneShot(choq_barra);
                boomeran.useGravity=true;
                Instantiate(explosion, contact.point, rota);
        }
        else{
            Instantiate(explosion, contact.point, rota);
            audio.PlayOneShot(exp_boo);
            Destroy (gameObject);
        }
}
```

Si ha chocado con las barras reproducimos el sonido, le aplicamos la gravedad al componente **Rigidbody** del boomerang para que así caiga, y creamos la explosión.

En el caso de que no haya chocado con las barras, se crea la explosión, se reproduce el sonido y se destruye el objeto, es decir, el boomerang.

El siguiente objeto es **Iniciar,** se encarga de poner todo en marcha, inicializando todas variables. Contiene un script llamado "**inicia_script.cs**".

Veamos la vista Inspector:



Tiene variables públicas en las cuales tiene los objetos en los cuales hay que notificar que comienza la partida.

```
void Start () {
    contador_iniciar.SendMessage("iniciar",SendMessageOptions.DontRequireReceiver);
    Instantiate(salir);
    Instantiate(ayuda);
}
```

La función Start manda el mensaje al contador para que comience la marcha atrás, crea el botón de salir y de ayuda.

```
void Update () {
    if((!GameObject.Find("boomeran_p(Clone)"))&&(!GameObject.Find("boton"))&&(!pausa)){
        boton_disparar.SendMessage("Activa_game",SendMessageOptions.DontRequireReceiver);
    }
}
```

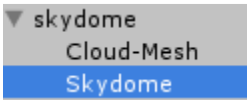
La función **Update** comprueba si no hay ningún boomerang realizando la trayectoria, si no está el botón activado y si no estamos en un momento de pausa para así activar el botón.

```
void pausa_up () {
    pausa=true;
}
void pausa_down () {
    pausa=false;
}
```

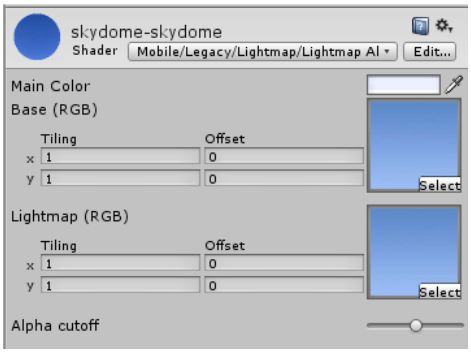
Y por último con estas funciones indicamos que estamos en un momento de pausa y que por lo tanto no debemos activar el botón de disparar.

El objeto **Point Light** solo tiene como función la de iluminar la escena, solo ha sido creado y no ha sufrido ninguna modificación.

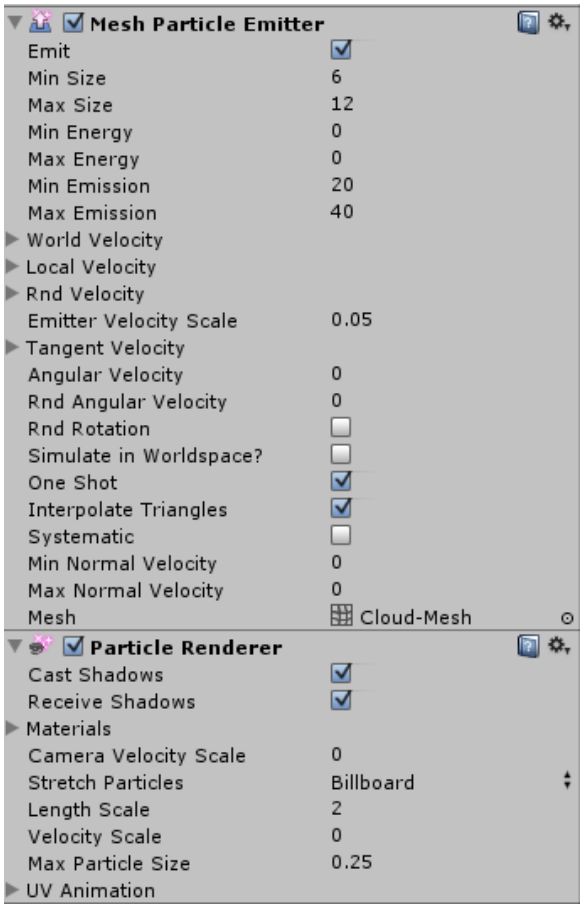
El siguiente objeto corresponde al cielo de la escena, me decidí por poner un skydome ya que un skybox hacía un cielo cuadriculado y además el skydome con las nubes daban un efecto espectacular.



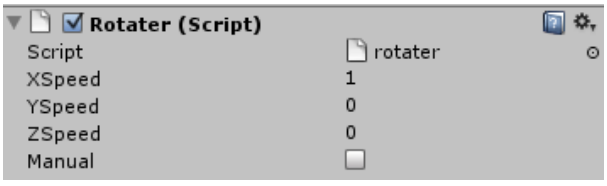
El objeto "skydome" se compone de objeto para emitir partículas que simula las nubes y una cúpula a la cual se le aplicado una textura azul para el cielo.



El Cloud-Mesh, tiene un script con el cual realiza el movimiento de girar las partículas para simular el movimiento de las nubes.



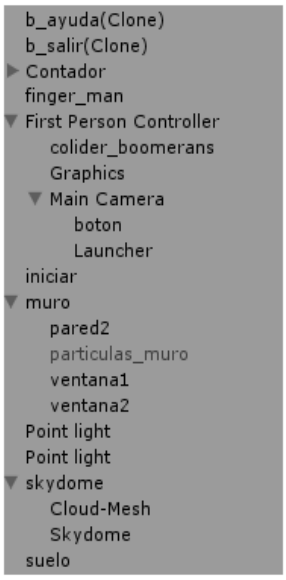
Como ya explicamos en el apartado de las clases, hay diversos atributos para configurar como emitir las partículas, y lo mejor en este caso es desde el **Inspector**.



El siguiente y último de la lista de la escena, es el **suelo**, que simplemente es un cubo (GameObject), al cual se le ha aplicado la textura del suelo, la misma que se aplicó al suelo del menú.

Escena del Muro "escena_pared2":

Veamos el esquema de objeto de la vista **Hierarchy** en el momento inicial de la escena:



Como se puede ver, hay muchos objetos en común con la anterior escena por lo que no entraremos a comentarlos de nuevo.

Estos objetos son:

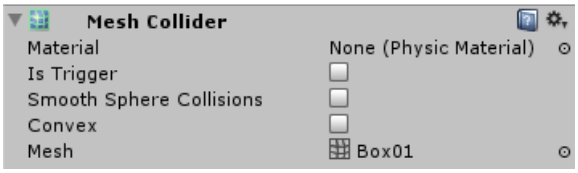
- **b_ayuda(Clone):** es el botón de ayuda de la esquina superior izquierda, y como ya comentamos en su momento, con una condicional comprobábamos en que escena estábamos y instanciábamos un menú de ayuda u otro.
- **b_salir(Clone):** es el botón de salir situado en la esquina superior derecha y al presionarlo se obtiene un menú de pausa.
- **Contador:** es el mismo para las tres pantallas y realiza una cuenta atrás de 60 segundos.
- **finger_man:** es el encargado de controlar las entradas de la clase Touch de la pantalla y notificarlo a los objetos.
- **First Person Controller:** el personaje es el mismo y el objeto a lanzar también es el mismo, por lo que el objeto para realizar estas funciones se repetirá en todas las escenas.
- **iniciar:** se encarga de inicializar las variables y los objetos para que pueda comenzar la partida. Además, controla cuando hay que habilitar el botón para que pueda disparar.

- **skydome:** es el cielo de la escena.
- **suelo:** es el suelo de la escena y es el mismo la escena del muro y boomerang.

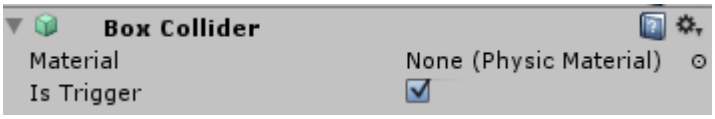
Ahora vamos a explicar el objeto "**muro**", es un objeto con dos ventanas las cuales debe atravesar el boomerang primero por la ventana de la derecha y luego por la de la izquierda para ir sumando y poder alcanzar el objetivo.

Contiene cuatro objetos hijos, los cuales son:

- **pared2:** este objeto se construyo con 3D Studio Max y luego importado a Unity, luego le añadimos el componente **collider** para que detectara las colisiones con el boomerang en caso de que se chocase con el muro.



- **particulas_muro**: este objeto es un sistema de partículas, se crea seleccionando GameObject->Create Other->Particle System, luego modifique los atributos de los componentes para cuando el muro comience a moverse realice el efecto de polvo que se levanta. En primera instancia, el objeto se encuentra deshabilitado y se habilitará cuando pase el primer boomerang por las dos ventanas y comience el muro a moverse.
- **ventana1**: tiene la misma función que la zona de barras, detectar cuando el boomerang atraviesa la ventana. Para crear este objeto, se crea un objeto vacio (GameObject->Create Empty) y le añadimos el componente box collider (Component->Physics->Box Collider) y seleccionamos la opción de trigger, para que los objetos puedan atravesar el objeto.



También se añade un script "ventana1.js" para insertar la funcionalidad que queremos añadir al objeto cuando sea atravesado.

```
function OnTriggerEnter(coll : Collider) {
    if (coll.tag=="boo"){
            bol=true;
            Debug.Log("ventana1 pasada");
    }
}
```

Cuando un objeto pasa por la ventana, es detectado con la función OnTriggerEnter, en ella miramos si el objeto de dicha colisión esta etiquetado como "boo", es decir, es el boomerang, en caso correcto la variable booleana "bol" se iguala a true.

- **ventana2:** es un objeto construido de la misma forma que ventana1, pero con distinta funcionalidad, por lo que tiene un script diferente "ventana2.js".

```
function Start () {
    cont=0;
}
```

Cada vez que se inicia la escena ponemos el contador a 0.

Cuando el boomerang pasa por la ventana 2, **OnTriggerEnter** registra la colisión, con la cual comprobamos si el objeto que ha colisionado es el boomerang, gracias a la etiqueta (tag) de este, y si antes había pasado por la ventena1. En caso correcto, se incrementa el contador en 1y notificamos al objeto muro, asociado a la variable "not_pared" que se ha incrementado el contador.

Una vez hayan pasado 3 veces por las ventanas, el objetivo se habrá cumplido y por lo tanto si instancia el objeto con el mensaje de conseguido (obj_conseguido), asignamos el contador, botón de salir y ayuda a variables para notificarles que el objetivo se ha cumplido.

El contador se para, el movimiento del personaje se deshabilita, se quita los botones de ayuda y salir, y se oculta el botón de disparar.

```
function OnTriggerEnter(coll : Collider) {
    if ((coll.tag=="boo")&&(ventana1.bol==true)){
                cont=cont + 1;
                not_pared.SendMessage("inc_cont",SendMessageOptions.DontRequireReceiver);
                Debug.Log("Boomeranes pasados"+cont);
                if (cont==3) {
                    Instantiate(obj_conseguido);
                    notificationObject=GameObject.Find("Contador");
                    boton_salir=GameObject.Find("b_salir(Clone)");
                    boton_ayuda=GameObject.Find("b_ayuda(Clone)");
                    notificationObject.SendMessage("parar",SendMessageOptions.DontRequireReceiver);
                    inicio.SendMessage("pausa_up",SendMessageOptions.DontRequireReceiver);
                    boton_fire.SendMessage("ocultar",SendMessageOptions.DontRequireReceiver);
                    boton_salir.SendMessage("quitar_boton",SendMessageOptions.DontRequireReceiver);
                    boton_ayuda.SendMessage("quitar_boton",SendMessageOptions.DontRequireReceiver);
                }
        }
}
```

Y por último, el objeto padre, **muro,** tiene incorporado un script para mover el muro y activar las partículas cuando se haya pasado como mínimo un boomerang por las ventanas.

El script se llama “mover_muro.cs”:

```
void Start () {
    div=2;
    cont=0;
    x=-1;
}
```

En la función Start, inicializamos las variables que serán usadas en las funciones **Update**() e **inc_cont**().

```
void inc_cont() {
    cont=cont+1;
}
```

El método inc_cont() es llamado cuando el boomerang pasa por las ventanas correctamente, de tal modo que, si el contando es igual a uno el muro comience a moverse y si es igual a dos duplique su velocidad.

Cuando el contador es igual a 1, es decir, el boomerang ya ha pasado una vez por las ventanas, se activa el sistema de partículas con la instrucción “gameObject.SetActiveRecursively” y comienza a mover el muro con la instrucción “transform.Translate”.

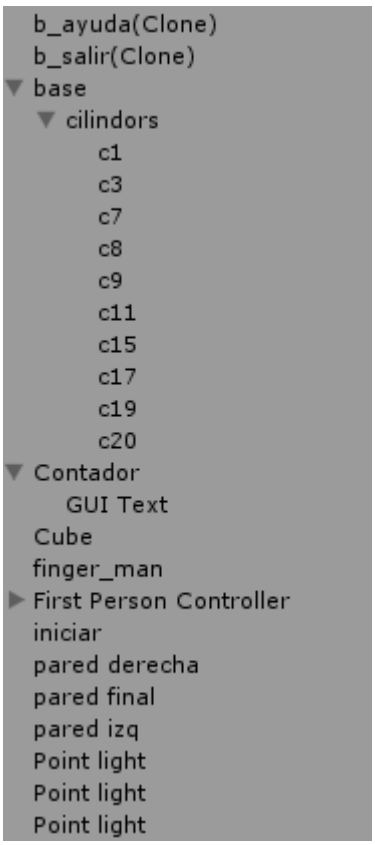
En principio el divisor (div) que se aplica a la fuerza es igual a dos, pero cuando el contador es igual a dos, el divisor se igual a 1 o -1 para duplicar la velocidad.

```
void Update () {
    if((cont==2)&&(div!=1)){
        if(div>0) {
            div=1;
        }else{
            div=-1;
        }

    }
    if((transform.position.x>248)&&(div>0)) {
        div=div*x;
    }
    if((transform.position.x<238)&&(div<0)) {
        div=div*x;
    }
    if(cont>=1){
        gameObject.SetActiveRecursively(true);
        transform.Translate(Vector3.right * Time.deltaTime/div, Space.World);
    }
}
```

Con la instrucción “transform.position.x” controlamos la posición del objeto para que el objeto se mueva dentro de unos límites.

Escena de Giratorio “esc_giratorio”:

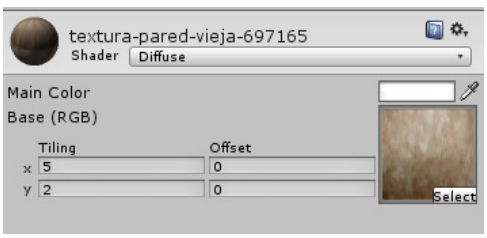
Veamos el esquema con los objetos nada más comenzar la escena en la vista Hierarchy:



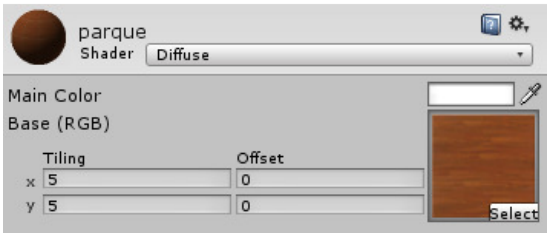
Como en la escena anterior, los objetos botón de ayuda, botón de salir, contador, finger_man, First Person Controller e iniciar, ya han sido explicados en lo que se refiere a su diseño e implementación.

Los objetos de Point light, puntos de luz, pared derecha, pared izq, pared final y cube, se corresponden con el escenario de la escena.

Las tres paredes tienen asignada la misma textura:

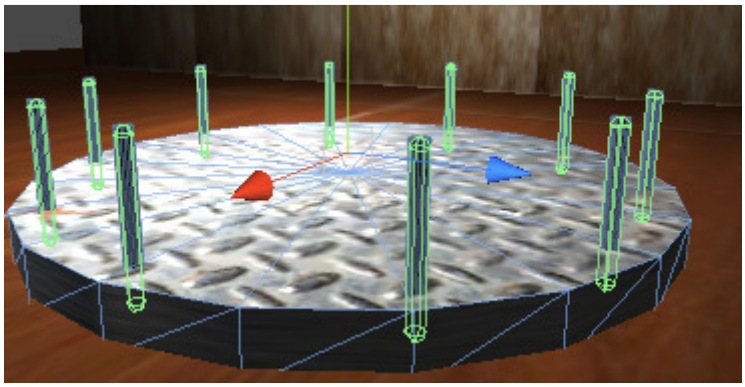


Y el suelo, objeto cube, tiene asignada una textura de parqué:



El principal objeto de esta escena es la base, es un objeto cilíndrico llamado base, sobre el cual se han colocado 10 cilindros. El objetivo de la pantalla es dar a todos los cilindros de la base para pasar la pantalla. Cuando se golpea un cilindro el objeto cilindro cambia de textura para marcar que ya está golpeado.

Tanto para crear la base como los cilindros seleccionamos GameObject->Create Other->Cylinder, pero variamos las características de ellos, ya que la base tiene menos altura y mucho más radio que los cilindros.



La base tiene asociado un script para que la base gire, este script se llama “girar_base.js”.

```
function Start() {
    cont_barras=0;
}
```

También tiene como función controlar el numero de barras que han sido golpeadas, y cuando se haya cumplido el objetivo crear los objetos necesarios.

En la función **Start** inicializa el contador de barras a 0, ya que esta función se ejecuta una vez solo cuando se inicia la escena.

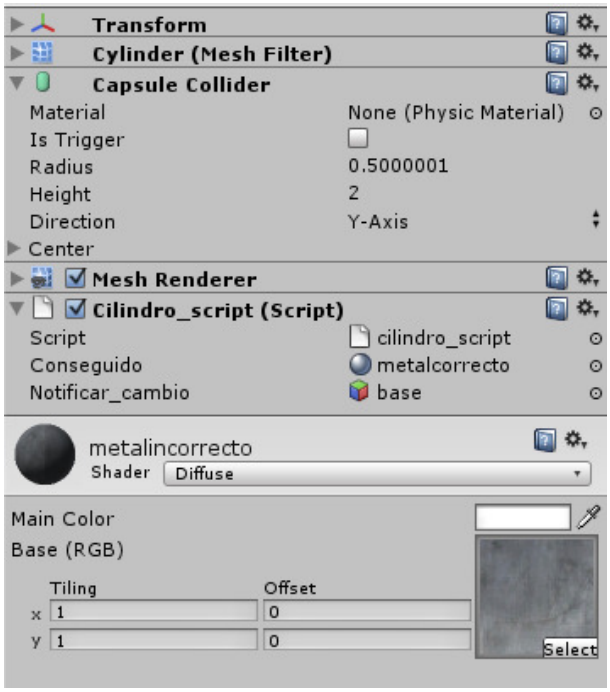
```
function Update () {
        if((cont_barras==10)&&(!GameObject.Find("conseguido(Clone)"))){
            Instantiate(conseguido);
            notificationObject=GameObject.Find("Contador");
            boton_salir=GameObject.Find("b_salir(Clone)");
            boton_ayuda=GameObject.Find("b_ayuda(Clone)");
            notificationObject.SendMessage("parar",SendMessageOptions.DontRequireReceiver);
            inicio.SendMessage("pausa_up",SendMessageOptions.DontRequireReceiver);
            boton_fire.SendMessage("ocultar",SendMessageOptions.DontRequireReceiver);
            boton_salir.SendMessage("quitar_boton",SendMessageOptions.DontRequireReceiver);
            boton_ayuda.SendMessage("quitar_boton",SendMessageOptions.DontRequireReceiver);
        }
        //Para que gire la plataforma
        transform.Rotate(Vector3.up * Time.deltaTime*-10, Space.World);
}
```

Cada frame se ejecuta la función Update, y cada vez que esto ocurre rotamos la base con la función "transform.Rotate". También comprobamos si el numero de barras golpeadas es igual a 10 lo cual significaría que se ha cumplido el objetivo, y comprobamos también que no hemos creado ya el objeto "conseguido (Clone)" para no estar cada frame creando uno cuando el objetivo se hubiera cumplido.

Cuando se cumple el objetivo, se instancia el objeto que porta el mensaje de conseguido y para el contador, oculta el botón de disparar y destruye los objetos de botón de ayuda y botón de salir.

```
function barra_activa(){
    cont_barras++;
}
```

Con la función incrementamos el contador de número de barras golpeadas. Esta función será llamada desde los cilindros cuando sean golpeados.



En cuanto a los cilindros, veamos que componentes tiene. Como se puede observar, tiene asignada una capsula para capturar las colisiones (Capsule Collider) y luego tratarlas en el script "cilindro_script.cs". El script posee dos variables públicas, "Conseguido" la cual tiene la textura a aplicar cuando el cilindro haya sido golpeado, y "notificar_cambio" la cual es la base, a la que hay que notificar que se ha cambiado de textura el cilindro para que incremente el contador.

Cuando el cilindro es golpeado, se captura la colisión, y comprobamos si no ha sido cambiado de textura, es decir, no se ha golpeado anteriormente, notificamos que se ha golpeado el cilindro, y ejecutamos el método **Cambio_textura().**

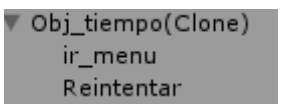
```
void OnCollisionEnter(Collision collision) {
    if(gameObject.renderer.sharedMaterial != conseguido) {
        notificar_cambio.SendMessage("barra_activa",SendMessageOptions.DontRequireReceiver);
        Cambio_textura();
    }
}
```

```
void Cambio_textura(){
    //cambiamos la textura a la correcta del metal
    gameObject.renderer.sharedMaterial = conseguido;
}
```

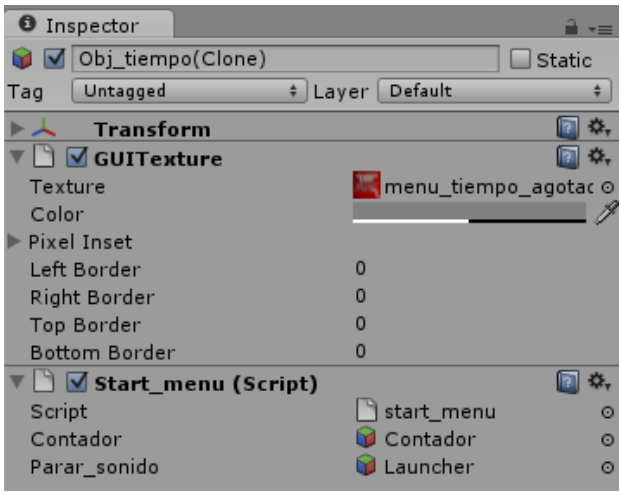
Este método tiene como función cambiar el material de textura, por una textura dorada almacenada en la variable conseguido.

Cuando el **tiempo** se acaba en cualquiera de las escenas de juego, se instancia un objeto como ya hemos hablado anteriormente. Este objeto es el siguiente:

Este objeto consiste de un objeto padre con la textura del mensaje, y dos objetos hijos con texturas intercambiables que realizan la función de botón.



El **Obj_tiempo(Clone)**, como se ve en la imagen siguiente, correspondiente a la vista Inspector, tiene asignada el componente GUITexture y la textura "menú_tiempo_agotado.jpg". Y contiene el script "start_menu.cs", con dos variables para cuando este objeto se cree, parar el contador y el sonido del disparo siempre y cuando se estuviera disparando cuando el tiempo se acabe.

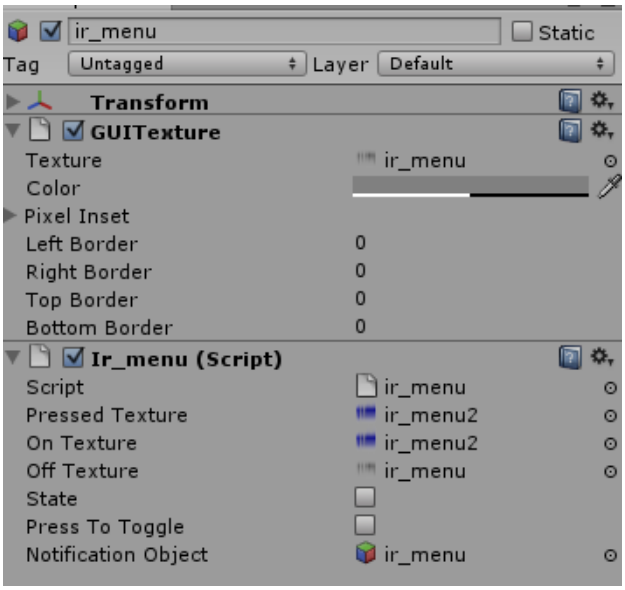


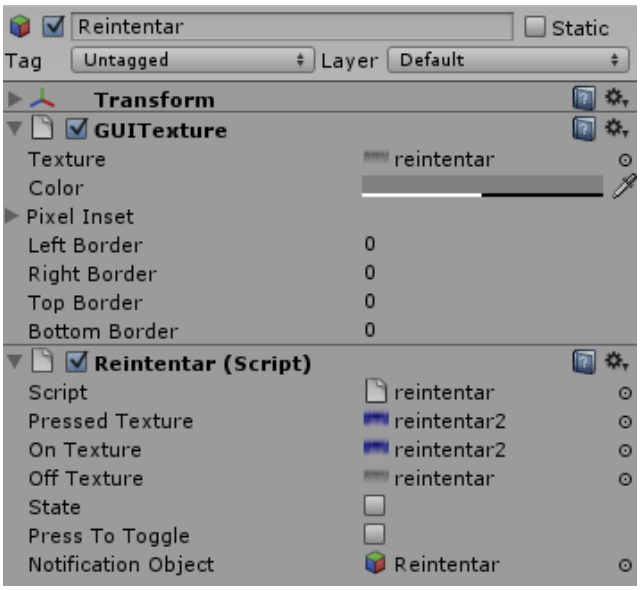
Cuando el objeto se instancia, hay que para el contador, el sonido, y destruir los botones de ayuda, salir y disparar.

```
void Start () {
    //PARAMOS EL CONTADOR,Y DESTRUIMOS BOTON DE AYUDA, SALIR Y DISPARAR
    contador=GameObject.Find("Contador");
    parar_sonido=GameObject.Find("Launcher");
    contador.SendMessage("parar",SendMessageOptions.DontRequireReceiver);
    parar_sonido.SendMessage("parar_sonido",SendMessageOptions.DontRequireReceiver);
    Destroy(GameObject.Find("b_ayuda(Clone)"));
    Destroy(GameObject.Find("b_salir(Clone)"));
    Destroy(GameObject.Find("boton"));
}
```

No es necesario luego crear los botones de nuevo ya que las opciones que ofrece el menú son "ir a menú" y "reintentar", las cuales cargan escenas directamente.

Veamos sus componentes en la vista Inspector:





Los dos objetos están construidos de forma similar, contiene un script que, como ya explicamos en los otros menus, intercambia las texturas cuando es presionada (touch). Estos eventos los controlan en la función Update(), y cargan las funciones doTouchDown y doTouchUp para cambiar de textura y cargar la escena correspondiente:

```
//COMPROBAMOS EN QUE ESCENA ESTAMOS PARA VOLVER A CARGARLA
if(GameObject.Find("barras"))
    Application.LoadLevel(2);
else if (GameObject.Find("muro")) {
    Application.LoadLevel(3);
}else
    Application.LoadLevel(4);
```

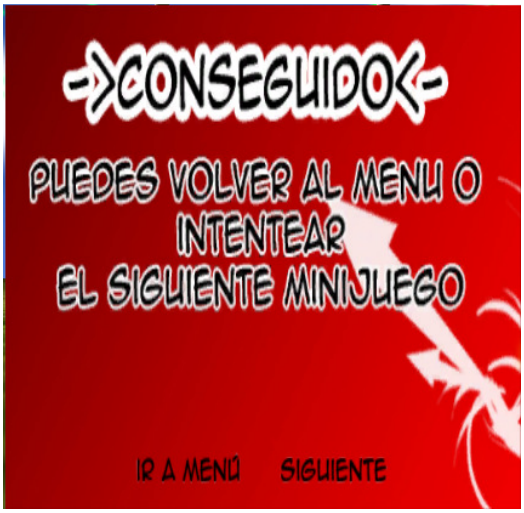
Función doTouchUp de “reintentar.cs”.

Como este menú, es el mismo en las tres pantallas, comprobamos en cual estamos y dependiendo de cuál sea, cargamos una escena u otra.

```
//CARGAMOS LA ESCENA MENU
Application.LoadLevel(0);
```

Aunque en las tres escenas se cargue el mismo menú, en este caso la finalidad es la misma y con una instrucción es suficiente.

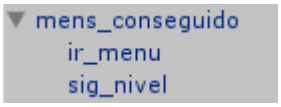
Cuando consigues el **objetivo** de las pantallas se crean un menú felicitándote por el logro y dando opción a acceder a los niveles superiores y acceder al menú, o, en el caso de la escena del giratorio, al ser el último nivel solo la opción del menú.

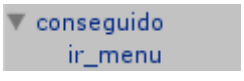


Este objeto está presente en las dos primeras escenas, del boomerang y el muro, ya que como se observa en la imagen posee las dos opciones.

Mientras que en la escena del giratorio solo aparece la opción de ir a menú.

Estos objetos han sido construidos de la misma forma que los demás menús. Veamos su estructura en la vista Hierarchy:





El primero corresponde al objeto con dos opciones y el segundo al objeto de la escena giratorio, cuando ya se han conseguido pasar todas las escenas.

La diferencia entre estos dos objetos es la textura del objeto padre, ya que cambia el mensaje a mostrar en los menús, y por supuesto que uno tiene un objeto hijo menos debido a las opciones.

Como se puede observar en las dos imágenes utilizan diferente textura.

Cuando el objeto **ir_menu** es pulsado se ejecuta la función **doTouchUp()** que contiene la instrucción que carga la escena del menú.

```
//aki cargar el menu dos.
Application.LoadLevel(0);
```

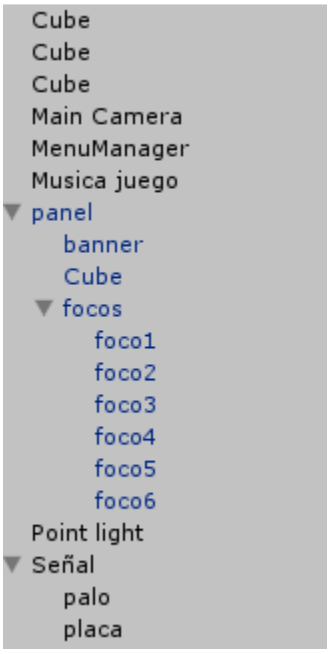
Esto se ejecuta en ambos menús, pero en el caso de **sig_nivel**, cuando es pulsado, al ejecutarse la función doTouchUp() se llevan a cabo las siguientes instrucciones:

```
if (GameObject.Find("barras"))
    Application.LoadLevel(3);
else
    Application.LoadLevel(4);
```

Comprobamos si no encontramos en la escena barras, para en ese caso, cargar la escena del muro, y si no cargar la escena del giratorio.

Escena de Créditos "creditos":

Veamos ahora el esquema de objetos de la vista Hierarchy (jerarquía) de la escena créditos.

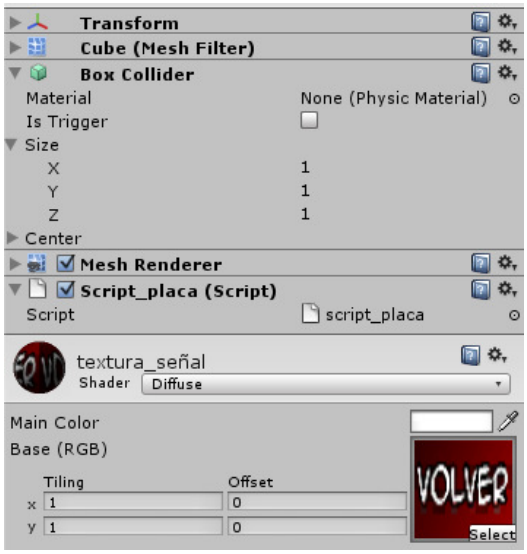


La escena comparte varios objetos ya explicados en las escenas del menú. Como son las tres paredes (cubes), Main Camera, MenuManager, Musica juego y el panel.

Es una escena informativa simplemente por lo que no tiene mucha complicación, como novedad la señal que permite volver al menú inicial.

La señal está compuesta por un palo y la placa, en la placa le hemos incorporado un script para que cuando sea pulsada comience a girar.

Veamos la vista **Inspector** de la placa:



La placa contiene Box Collider para detectar las colisiones y poder ser tratadas en el script.

El script al iniciar el objeto con la función Start() pone la variable booleana “pulsada” a false indicando que principio no está pulsada la placa.

En la función Update cuando la variable “pulsada” está activada se le aplica a la placa la rotación mediante la función de la clase “Transform” **Rotate.**

Las siguientes funciones se implementan para recibir las señales enviades desde el objeto MenuManager que posee el script FingerManager.cs que ya hemos explicado. De este modo, cuando la placa comienza a ser pulsada se registra con la función FingerBegin y activa la variable booleana.

Y cuando la placa deja de ser pulsada se ejecuta la función **FingerCancel** o **FingerEnd**, en las cuales se desactiva el giro poniendo a false la variable booleana y se carga la escena del menú.

```
public class script_placa : MonoBehaviour {
    static bool pulsada;
    // Use this for initialization
    void Start () {
        pulsada=false;
    }

    // Update is called once per frame
    void Update () {
        if(pulsada)
            transform.Rotate(Vector3.up * Time.deltaTime*-150, Space.World);
    }

    void FingerBegin (iPhoneTouch evt) {
        pulsada=true;
    }

    void FingerMove (iPhoneTouch evt) {
    }

    void FingerEnd (iPhoneTouch evt) {
        pulsada=false;
        Application.LoadLevel(0);
    }

    void FingerCancel (iPhoneTouch evt) {
        pulsada=false;
        Application.LoadLevel(0);
    }
}
```

3.5.Resultado:

Despues de toda la implementación del proyecto, conseguimos que la aplicación corriese en un iPad con alguna que otra dificultad, que ya trataremos más adelante.

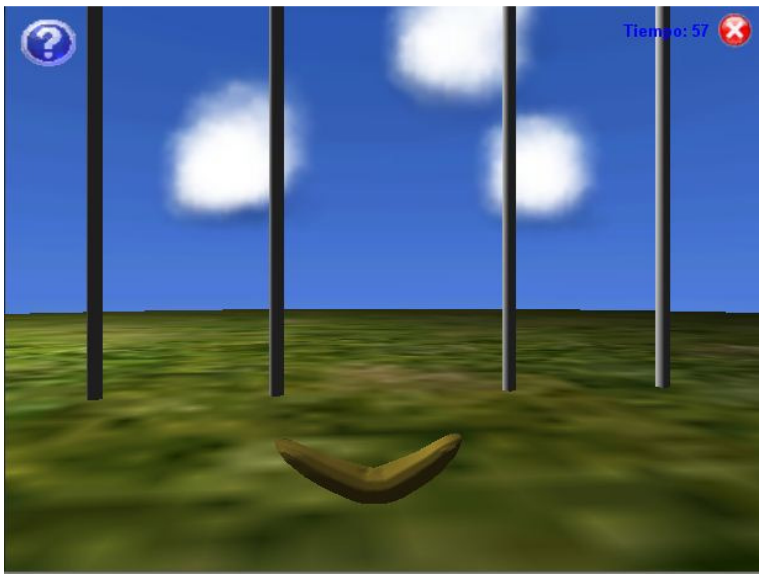
Veamos una serie de pantallazos de las escenas en los cuales podemos ver que gran parte de los requisitos, el resto, se observan mientras se practica el juego:

"Escena del menú"

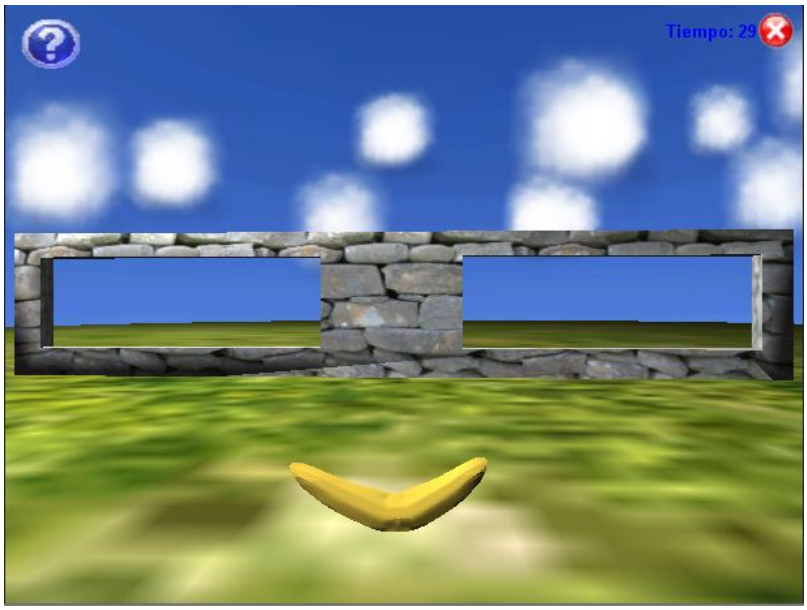
Se pueden observar las tres opciones nombradas en los requisitos, y además un panel informativo el cual se ilumina cuando es pulsado.

"Escena del menú dos"

Se ven las cuatro opciones del menú para acceder a los diferentes niveles, y regresar al menú principal. Además también posee el panel informativo.



"*Escena boomerang"* Se observa en la imagen los botones ayuda y de salir, el contador de tiempo y el botón para disparar el boomerang. Se ve también las barras por entre las cuales tiene que pasar el boomerang en la prueba.



"*Escena del muro"* Se obsrva como en la anterior, los botones de ayuda y de salir, y el boton de disparar. Se ve el muro con las ventanas por las que debe de pasar el boomerang, y el contador de tiempo indicando el tiempo que resta para concluir la partida.

"Escena del giratorio" Como se ve en la imagen, posee los botones de salir y ayuda, así como el de disparar. Se ve la base con los cilindros objetivo los cuales hay que golpear y el tiempo que falta para terminar la partida.

"*Escena de créditos"* Es una escena informativa, como se puede ver en la imagen, en la cual sale el logo del juego y el nombre del creador sobre el panel, en esta imagen se puede observar también que el panel se ilumina al presionar sobre el. En el margen inferior izquierdo se ve la señal que nos da opción a regresar al menú principal.

3.6.Problemas encontrados:

Al crear una aplicación sobre aplicaciones que en principio no se conocen, como Unity 3D, sobre dispositivos los cuales no conoces sus características como el iPad suelen provocar varios problemas que vamos a comentar en este apartado.

Uno de los principales problemas era adaptar nuestra aplicación para que funcionase de manera correcta en el iPad. En un principio al crear la aplicación, el tema de los menús y modos de disparar el boomerang era por teclado y ratón. Este tipo de entrada tiene una clase diferente a la clase que controla la entrada de los iPad, iPhoneInput, en especial iPhoneInput.touches. Al haber instalado la aplicación Unity sobre Windows, la documentación sobre este tipo de entrada no se instala por defecto.

Por lo tanto, al tener que adaptar toda la aplicación para su funcionamiento sobre el iPad hizo que tuviera que modificar gran parte del código de está.

Otro problema de adaptación a iPad era que, como no dispongo del dispositivo, no podía testear que el código que modificaba era correcto o no, más que en tutorías, lo cual ralentizaba el proceso de construcción.

Estos problemas los solucionamos con un paquete que importamos en el proyecto en Windows que simulaba las entradas de iPad en lo referente a toques (touches) mediante el puntero del ratón. De esta forma ya podía comprobar el funcionamiento correcto del sistema en lo táctil.

El tema del acelerómetro no se podía simular, por lo que las pruebas las realizamos en tutorías comprobando los valores que tomaba el dispositivo para controlar el movimiento. En un principio el movimiento era muy rápido, lo cual hizo que modificáramos los valores de velocidad con respecto al movimiento del dispositivo.

En posteriores sesiones de tutorías, al probar el juego, descubrimos que a la menor inclinación el personaje del juego se movía. Esto se debía solucionar ya que es muy difícil mantener sobre las manos el dispositivo completamente recto. El valor de entrada del acelerómetro toma valores del -1 al 1, lo cual decidimos que mientras que el valor no superase 0.45 o, en caso negativo, no bajara de -0.45, no le aplicaríamos movimiento al personaje. Con las posteriores pruebas comprobamos que 0.45 equivalía más o menos a 45º de inclinación.

El mayor de los problemas, vino cuando ejecutábamos la aplicación sobre el iPad y no corría de manera correcta, iba demasiado despacio. Insertamos parte de código a la aplicación para que mostrase la cantidad de frames por segundo a la que se ejecutaba la aplicación y vimos que era infinitamente inferior la del iPad a la de un ordenador, pasaba de 80 frames por segundo del ordenador a 8 o 10 frames por segundo. La

sensación de lentitud la asociamos a que no era capaz de ejecutar más de 24 frames por segundo, como en cine.

Después de investigar pos distintos foros, asociamos que el renderizado de las figuras de las escenas era el causante de este problema. La cantidad de vértices que contienen las figuras de la escena hace que el iPad no pueda renderizarlos con la velocidad adecuada.

En los foros dicen que un iPad puede renderizar alrededor de 40K vértices por frame, y por ejemplo, la malla del objeto boomerang tiene 809 vértices.

Es por ese el motivo por el cual la aplicación se ejecuta correctamente en un PC y no en un iPad, por la escasa potencia gráfica del dispositivo.

4. Conclusiones y líneas futuras:

4.1. Conclusiones técnicas:

Vamos a estudiar en este apartado si se han cumplido los objetivos del proyecto y si la planificación inicial de desarrollo de éste ha sido alterada.

Estos puntos eran los objetivos del proyecto:

- Estudio de las herramientas, lenguajes y programas a utilizar en el diseño de la aplicación.
- Diseño de la aplicación.
- Implementación de la aplicación.
- Pruebas de la aplicación.
- Documentación de la aplicación para el correcto manejo del interface.

Para realizar el estudio de las herramientas, leí tutoriales básicos sobre el funcionamiento del interfaz de Unity, y luego realicé dos tutoriales de la web de Unity 3D, uno sobre un coche (http://unity3d.com/support/resources/tutorials/car-tutorial) y otro sobre un shooter, que me facilitó el tutor por correo. Son tutoriales muy bien explicados que te ayudan a familiarizarte con las características y facilidades del programa.

El aprendizaje del lenguaje de programación C#, fue gracias a la documentación de la web www.unity3d.com y el foro español de la aplicación http://www.unityspain.com/ . Hay que apuntar que las grandes similitudes entre los lenguajes C# y Java facilitaron mucho el aprendizaje.

Los dos programas externos utilizados para la creación de esta aplicación han sido Adobe Photoshop para el tratamiento de texturas y 3D Studio Max para la creación de los modelos 3D. En el caso de Photoshop, ya poseía conocimientos adquiridos de forma autónoma. Y los conocimientos sobre 3D Studio Max fueron adquiridos en la asignatura “Gráficos y multimedia” de la Ingeniería Técnica Informática de Gestión cursada en la Universidad Pública de Navarra (UpNa).

En cuanto al diseño de la aplicación, comenzó principalmente con la idea de lanzar un boomerang y superar una serie de pruebas. Conforme iban pasando las sesiones de tutorías se fueron estableciendo y modificando los requisitos. Se realizaron los diagramas de estado ya que era la forma más precisa de explicar el funcionamiento de las escenas y finalmente se obtuvieron los resultados.

La implementación de la aplicación fue costosa por la constante variación de requisitos, migración de gran parte del código de Java Script a C#, ya que todos los tutoriales estaban programados en Java Scritp. Y por su puesto, la adaptación de la

aplicación para su funcionamiento sobre un iPad, como ya hemos comentado anteriormente. Pero finalmente se consiguió su correcto funcionamiento.

Las pruebas las realizamos conforme se iban obteniendo pequeñas metas, para ver su correcto funcionamiento y poder continuar implementando.

En cuanto a la documentación de la aplicación para su correcto manejo, no hemos incorporado un manual de usuario, pese a ser nuestra intención primaria, ya que el manejo de la aplicación es muy intuitivo, y la aplicación dispone de botón de ayuda para explicar los objetivos de las pruebas y como funciona, en caso de duda.

Veamos ahora la planificación inicial del proyecto:

PLANIFICACIÓN			
Nombre Etapa	**Fecha comienzo**	**Fecha fin**	**Duración**
Formación	21/10/2010	21/11/2010	1 mes
Iteración 1	21/11/2010	21/01/2011	2 meses
Iteración 2	21/01/2011	21/02/2011	1 mes
Iteración 3	21/02/2011	21/03/2011	1 mes
Iteración 4	21/03/2011	21/04/2011	1 mes
Documentación y memoria	21/04/2011	21/05/2011	1 mes

La etapa de **formación** para la creación del proyecto fue más costosa de lo que se planificaba debido a que se realizaron dos tutoriales completos. Finalmente esta etapa concluyó en diciembre, un mes más tarde de lo previsto.

La **iteración 1**, fue la más duradera ya que donde se comenzó a construir la aplicación, y fue más costosa en las fase de análisis y diseño, así como en la **iteración 2**.

La **iteración 3**, prácticamente toda se dedicó a la implementación de la aplicación.

Y en la **iteración 4**, se realizaron todas las pruebas, cambios en la implementación por fallos obtenidos en las pruebas, y se comenzó a documentar ya el grueso del proyecto, la implementación.

La fase de **documentación y memoria**, se cumplió el plazo establecido, ya que no se ha durado más de un mes en redactar la memoria, gracias a la documentación ya redactada anteriormente.

RESULTADO FINAL			
Nombre Etapa	**Fecha comienzo**	**Fecha fin**	**Duración**
Formación	21/10/2010	21/12/2010	2 meses
Iteración 1	21/12/2010	21/02/2011	2 meses
Iteración 2	21/02/2011	21/03/2011	1 mes
Iteración 3	21/03/2011	21/04/2011	1 mes
Iteración 4	21/04/2011	21/05/2011	1 mes
Documentación y memoria	21/05/2011	21/06/2011	1 mes

4.2. Conclusiones personales:

El realizar este proyecto de una aplicación para los dispositivos iPad da una visión general de las opciones que tiene esta parte del mercado de la informática.

Asimismo, el haber realizado este proyecto con la aplicación Unity 3D proporciona una gran cantidad de posibilidades en cuanto a la creación no solo de aplicaciones para dispositivos móviles, si no para PC, para la web, para videoconsolas como Play Station, Wii y Xbox. Se pueden crear aplicaciones no muy complejas para dispositivos móviles o tablets, que no requerirán mucho esfuerzo y obtendrás grandes resultados.

Este tipo de aplicaciones para iPad, pueden ser promocionadas en el AppStore, un almacen de aplicaciones, donde pueden ser vendidas y obtener grandes beneficios, siempre y cuando tengas un poco de suerte.

El hecho de haber realizado la aplicación desde cero hasta su implantación, ha permitido conocer la cantidad de problemas que te pueden surgir durante la construcción del producto y como dar solución a ellos. La acción de planificar y diseñar todos los pasos para crear el proyecto es muy importante para la correcta implementación de este.

Los conocimientos adquiridos durante la carrera, me han servido para analizar, planificar, diseñar, y, sobre todo, implementar la aplicación. Pese a tener algunos conocimientos previos adquiridos en el grado superior de Administracion de Sistemas Informáticos, la formación recibida durante la titulación me ha sido muy útil para los diferentes aspectos de creación de este proyecto, especialmente en la programación en C# debido a las similitudes con el lenguaje cursado en la carrera Java.

También, se han adquirido multitud de conocimientos en cuanto a la programación en C#, del cual no poseía previamente, y en lo referente a la aplicación Unity 3D, la cual ofrece una gran cantidad de facilidades para la creación aplicaciones, principalmente a la programación en tres dimensiones para crear juegos.

Finalmente quisiera agradecer la ayuda recibida a mi tutor de proyecto, Oscar Ardaiz, el cual ha ido resolviendo cualquier tipo de duda o problema surgidos.

4.3. Líneas futuras:

Una vez concluido el proyecto vamos a explicar dos aspectos que deberían ser tratados en un futuro para mejorar la aplicación.

El principal tema a tratar en un futuro sería solucionar el problema del renderizado de los objetos 3D en el iPad para que la aplicación se ejecutara correctamente. El problema es que el iPad no tiene capacidad para renderizar por cada frame todos los objetos de la escena.

Optimizar la geometría del modelo

Hay que tener en cuenta que el número real de vértices que el hardware de gráficos tiene que procesar no suele ser el mismo que lo que se muestra en una aplicación 3D.Las aplicaciones de modelado suelen mostrar el número de vértices geométricamente, es decir, número de puntos que componen un modelo.

Como objetivo hay que tener 10K vértices o menos por imagen.

Las texturas deben ser comprimidas, aun que reduzca mucho la calidad gráfica, también incrementa mucho el rendimiento. Evitar siempre que sea posible el tema de las sombras ya que ralentiza mucho el rendimiento, y las sombras no es algo funcionalmente necesario.

La versión que permite Unity descargar también tiene sus limitaciones. Unity Pro permite seleccionar objetos de las escenas como “static” indicando de esta forma que el objeto va a permanecer estático durante la ejecución de la misma, y de este modo solo renderiza el objeto una vez. Pero la versión Unity básica no da esta opción, si que permite seleccionar los objetos como estáticos paro a la hora de construir el proyecto solo deja la opción “Dynamic”.

Observar la siguiente imagen de la ventana Player Settings:

La segunda línea a tratar para el futuro sería la de promocionar la aplicación para venderla al **App Store.**

App Store es un servicio para el iPhone, el iPod Touch, el iPad, Mac OS X Snow Leopard y Mac OS X Lion, creado por Apple Inc., que permite a los usuarios buscar y descargar aplicaciones informáticas de iTunes Store o Mac App Store en el caso de Mac OSX, desarrolladas con el iPhone SDK y publicadas por Apple. Estas aplicaciones están disponibles para ser compradas o libres de costo, dependiendo de cada una. Las aplicaciones pueden ser descargadas directamente al iPhone o al iPod Touch por medio de una aplicación del mismo nombre, aunque App Store también está disponible al interior del programa informático iTunes.

Si bien Apple ha manifestado que no espera obtener ganancias de la tienda, Piper Jaffray predijo que App Store podía crear un mercado rentable con ingresos que excedan los mil millones de dólares anuales para la compañía. Apple otorga el 70% de los ingresos de la tienda directamente al vendedor de la aplicación y el 30% corresponde a Apple.

App Store fue inaugurada el 10 de julio de 2008 por medio de una actualización de iTunes. Las aplicaciones estuvieron inmediatamente disponibles para ser descargadas; sin embargo, la versión 2.0 del software del iPhone y el iPod Touch compatible con la nueva tienda aún no estaba disponible para ser descargada desde Apple Software Update, lo que provocó que las aplicaciones no pudieran ser instaladas. La versión 2.0 del iPhone OS fue lanzada el 11 de julio de 2008 y las aplicaciones ya pudieron ser transferidas a los dispositivos actualizados. Para el 8 de junio de 2009, ya existían más de 50.000 aplicaciones de terceros oficialmente disponibles para el iPhone y el iPod Touch en App Store. En menos de un año desde su lanzamiento, App Store superó los mil millones de descargas de aplicaciones.

Aplicaciones:

El 10 de julio de 2008, el Director ejecutivo de Apple, Steve Jobs declaró a USA Today que la App Store contenía 500 aplicaciones de terceros para iPhone y iPod Touch, 125 de las cuales eran gratuitas. Estas aplicaciones de terceros variaban desde aplicaciones para negocios, juegos, entretenimiento, educativas y muchas más. Para el 11 de julio de 2008, los usuarios podían comprar aplicaciones de la App Store y transferirlas al iPhone o el iPod Touch con la actualización de software iPhone 2.0 que estuvo disponible a través de iTunes ese mismo día. El primer fin de semana fueron descargadas 10 millones de aplicaciones.

El 16 de enero de 2009, Apple anunció en su página web que 500 millones de aplicaciones habían sido descargadas. El 23 de abril de 2009, un niño de 13 años de edad, Connor Mulcahey, de Weston, Connecticut, alcanzó la cifra de mil millones de aplicaciones descargadas.8 El 22 de enero de 2011 se descargó la aplicación número diez mil millones.

Clasificación de las aplicaciones:

A continuación se listan las categorías de clasificación de Apple:

Clasificación	Descripción
4+	No contiene material desagradable.
9+	Puede contener situaciones leves o infrecuentes de violencia realista, fantástica o en dibujos animados, y contenido sugestivo, maduro o de terror que puede no ser apropiado para menores de 9 años.
12+	Puede contener lenguaje no apropiado leve o infrecuente, violencia realista, fantástica o en dibujos animados frecuente, y contenido maduro o sugestivo leve o no frecuente, y juegos de azar simulados que pueden no ser apropiados para menores de 12 años.
17+	Puede incluir contenido maduro, sugestivo o de terror intenso y frecuente; más contenido sexual o de desnudez, alcohol, tabaco, y drogas que puede no ser apropiado para menores de 17 años. Los consumidores deben tener al menos 17 años para comprar aplicaciones con esta clasificación. Siempre que una aplicación con esta clasificación sea descargada, se mostrará un mensaje preguntando si el usuario tiene 17 años o más.

Apple clasifica las aplicaciones en base a su contenido, y para cada una determina para que grupo de edad es apropiada. Según el evento de lanzamiento del iPhone OS 3.0, el iPhone permitirá bloquear las aplicaciones desagradables en las opciones de éste.

Cómo promocionar aplicaciones en la App Store

App Store es un método conveniente para desarrolladores de software para vender sus productos en una plataforma universal. Luego de tener sus apps aprobadas, los desarrolladores pueden distribuir su software a muchas computadoras equipadas con App Store. El paso más difícil en este proceso es promover la aplicación (o app) en sí y generar un gran número de ventas. Muchos métodos disponibles pueden incrementar la popularidad de una app bien construida.

Pasos para promocionar una app en la App Store

- **Paso 1.** Considera iniciar un blog para tu app para crear un nombre para tus apps y para ti mismo como desarrollador. Las personas se interesarán en los productos que ofreces. Eventualmente desarrollarás un gran número de seguidores que se mostrarán animados en estar en el "Opening Day" (Día de Lanzamiento) de tu próxima app. Los blogs también incrementan la exposición de tus apps en herramientas de búsquedas populares.
- **Paso 2.** Registra tus apps en sitios de comentarios y reseñas de apps. Muchos sitios se especializan en comentar sobre apps que se encuentran disponibles en el mercado (marketplace). Registra tu app a los sitios Web de reseñas que encuentres. Hasta una reseña bien escrita puede impulsar a miles de descargas de tu aplicación. Existen muchos sitios de reseñas populares incluyendo: TouchArcade.com, MacAppStoreReview.com y AppStorm.net.
- **Paso 3.** Distribuye vales de descuento para tu app en tabloides de mensajes y sitios de redes sociales. Los desarrolladores tienen la habilidad de crear vales de descuento que actúan como certificados de descargas gratuitas para propósitos promocionales. Cada vale puede ser canjeado en línea por una descarga gratis de tu software. Este es especialmente útil al registrar apps a sitios para reseñas y comentarios que posiblemente no desean pagar inicialmente por tu app.

5. Bibliografía:

En este apartado vamos a ver los libros, paginas web, etc que se han tenido como referencia para el aprendizaje y creación de la aplicación de nuestro proyecto.

5.1.Libros de texto:

- MICROSOFT C#: LENGUAJE Y APLICACIONES de Ceballos, Francisco Javier
- LA BIBLIA DE C# - ANAYA – Brian Patterson, Pierre Boutquin y Meeta Gupta

5.2.Direcciones Web:

Web oficial de Unity 3D:

- www.unity3d.com

Foro de consulta español sobre la aplicación Unity3D:

- www.unityspain.com

Wiki inglesa sobre Unity 3D:

- http://www.unifycommunity.com/

Página de descarga de modelos 3D:

- http://www.turbosquid.com/3d-models/free-max-mode-boomerang-aboriginal-gaming/273053

Web de descarga de sonidos:

- http://efectos-de-sonido.anuncios-radio.com/gratis/index.php?option=com_weblinks&catid=71&Itemid=4
- http://www.pacdv.com/sounds/mechanical_sounds.html